Final Report
CS 3710: Computer Design

Ben Holt
Justin Olson
Usit Duongsaa
Russ Christensen
# Table of Contents

System Description .......................................................................................................................... 5
Design Description .......................................................................................................................... 6
  Top-Level ....................................................................................................................................... 6
    Why A Five-Stage Pipeline? ........................................................................................................ 6
Register File ..................................................................................................................................... 8
Instruction Memory ......................................................................................................................... 8
Data Memory .................................................................................................................................... 8
I/O .................................................................................................................................................. 8
Control Signals ............................................................................................................................... 9
  Signals Description ....................................................................................................................... 11
Clock Frequency ............................................................................................................................. 13
ALU Design .................................................................................................................................... 15
Op Stage Design ............................................................................................................................. 16
Register File Design ....................................................................................................................... 17
Data Memory .................................................................................................................................... 19
Decoder Design .............................................................................................................................. 20
  Instruction Decode Table ............................................................................................................. 22
I/O Interface Design ....................................................................................................................... 23
  VGA Controller .......................................................................................................................... 24
  Sound Device ............................................................................................................................. 27
  NES Game pad Controller .......................................................................................................... 28
Tool Set ......................................................................................................................................... 30
Tool Set ......................................................................................................................................... 30
Testing Framework ......................................................................................................................... 31
Programming Manual ................................................................................................................... 34
Operation Manual .......................................................................................................................... 37
The CASM Compiler/Assembler ..................................................................................................... 39
Show-Off Applications .................................................................................................................. 43
  Tetris ........................................................................................................................................... 43
  Space Invaders ........................................................................................................................... 43
Conclusions ..................................................................................................................................... 44
Appendixes ..................................................................................................................................... 47
VHDL Source Code ......................................................................................................................... 47
  ALU_16.vhd ................................................................................................................................. 47
  pipeline_synthesis.vhd ................................................................................................................ 50
VHDL TestBench Programs ............................................................................................................ 65
  test_alu.vhd ............................................................................................................................... 65
CASM Test Code ............................................................................................................................ 73
  test_jal.cpp ............................................................................................................................... 73
  test_scond_ge.cpp ...................................................................................................................... 74
  fib_recur.cpp ............................................................................................................................ 75
Show Off Applications Source Code ............................................................................................. 77
tetris.cpp.................................................................................................................... 77
invader.cpp.............................................................................................................. 119
System Description

Our Console Revolution 16 (CR16) was envisioned to be a game console similar in abilities to systems such as the original Nintendo or the Nintendo Gameboy. To achieve this goal, we designed an onboard VGA controller to output graphics. We have also designed and built an off board audio circuit capable of producing simple tones. For input we built a circuit to interface with an original Nintendo game pad.

From the beginning we designed our system to be fast enough to be a quality gaming console. Our goal was to have a system fast enough that a game like Space Invaders or Tetris would be able to run at 60 frames per second. We have reached this goal. At professor Erik Brunvand’s suggestion our processor is five stage pipelined.

Memory bandwidth is our speed limiting factor. The Flash EEPROM, which we use for instruction memory, limits our processor to one instruction per cycle at 5 Mhz. According to the Xilinx Webpack all other parts of our system are capable of running at 67 Mhz. We are able to achieve full bandwidth out of our memory system because of our pipeline. At all times we are reading in program memory at full speed. To achieve this we have two completely separate memory busses. We use the FlashEPROM for program memory, six BlockSRAM units for data, and 2 BlockSRAM units for video memory. These separate busses allow us to do many memory accesses in parallel which allows our console to run faster.
Design Description

Top-Level
Our process is a simple 5-stage pipeline.
- Fetch : Using the current PC, fetch instruction from memory
- Decode : Compute all the control signals for the fetched instruction
- Read : Read values off the Register File / Memory
- Op : Do operation/calculation on the data
- Write Back : Write result back to Register File / Memory

Why A Five-Stage Pipeline?
During one of our weekly meetings Professor Erik Brunvand suggested that we consider combining the Read and Op stages into one stage. The argument to combine the stages was that because the instruction set we use either reads from a register file and does some operation, or reads from memory and does not do any operation. The idea points out that register reading is very fast and we can do both reg-read and op in just one cycle. That a memory read might take longer, but because we will not do any operations, we will still need only one clock cycle for both the Read and Op stages.
After looking into this suggestion, we decided to stay with our five stage design for the following reasons:

1) We feel it is a cleaner design that is easier to implement and more importantly much easier to test if we separate the process of data input from the computation over that data.

2) Keeping our current design gives us the flexibility to add instructions that both read from memory and perform some operations. Given our goal to create a game console, there was a high chance that we would need to add instructions that did this in order to produce high-speed animated graphics. The Nintendo Gameboy is a good example. If we decide to extend our instruction set with graphics instructions similar to the way Nintendo extended the Z80 instruction set with their design of the Nintendo Gameboy then we would require the ability this ability to fully utilize both the Read and Op stage.

Now that we have the basic pipeline setup, we add memory and I/O.

Pipeline, Memory and I/O Architecture

As illustrated above, some components are used by more than one stage. This creates trouble, because components like the Register File must be unique, i.e. we cannot
have 2 register files, one for the READ and one for the WRITEBACK. They must be the same register file. We have different solutions for different components.

**Register File**

In the worst case, we need to be able to read 2 register values for the READ stage, and write one register for the WRITEBACK stage. Unlike a non-pipelined processor, the addresses of the three register can be totally different because the writing is from an instruction in the WRITEBACK stage, and the readings are from a different instruction in the READ stage. As a result we build a 3-port register file instead of 2 as specified in Lab 2.

**Instruction Memory**

We store instructions in a completely different unit (in the FLASH instead of SRAM) so we can fetch an instruction and perform a memory read/write at the same time. If we did not do this, we will need to stall the pipeline every time we do a memory operation, which is highly disastrous for memory-intensive programs like graphics and games.

**Data Memory**

We have the same problem as in the Register File. Namely, we might need to read one memory location for the READ stage, and write to a different location for the WRITEBACK stage. We realize later that this could have been accomodated using the dual-port feature of the BlockRAM. Instead we choose to simply prohibit memory reading and writing from occurring at the same cycle.

**I/O**

We design the I/O components in such a way so that a port is either read-only or write-only. This way, we can think of having 2 separate sets of I/Os, read-only components for the READ stage, and write-only components for the WRITEBACK
stage. We ended up using more address space this way, but given the 65,536 addresses we have, this is not an issue.

*** Each component will be explained in greater details later. ***

**Control Signals**

The diagram in the next page shows all control signals in the pipeline. The signals are generated in the decode stage and passed on to any component that needs it.
## Signals Description

The number in parenthesis indicates the width of the signal (number of bits).

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC (16)</td>
<td>The program counter of the current instruction</td>
</tr>
<tr>
<td>PC+1 (16)</td>
<td>The program counter of the next instruction</td>
</tr>
<tr>
<td>Instruction (16)</td>
<td>The current instruction</td>
</tr>
<tr>
<td>Immediate (8)</td>
<td>The immediate value field of the current instruction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectRegA (4)</td>
<td>Selects which register (0..15) to read as argument A</td>
</tr>
<tr>
<td>SelectRegB (4)</td>
<td>Selects which register (0..15) to read as argument B</td>
</tr>
<tr>
<td>RegA (16)</td>
<td>The value of the register read and passed as argument A</td>
</tr>
<tr>
<td>RegB (16)</td>
<td>The value of the register read and passed as argument B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWE (1)</td>
<td>The WriteEnable signal for the Register File</td>
</tr>
<tr>
<td>RegWrtSel (4)</td>
<td>Selects which register (0..15) to write to</td>
</tr>
<tr>
<td>PCResSel (1)</td>
<td>Indicates what data is to be written to the Register File come from 0 PC, 1 Result of OP stage</td>
</tr>
<tr>
<td>RegWrtData (16)</td>
<td>The data to write to the register file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp (2)</td>
<td>The operation the ALU will do (add/and/or/xor)</td>
</tr>
<tr>
<td>isNegate (1)</td>
<td>Whether the 2\textsuperscript{nd} argument to the ALU should be negated before doing any operation</td>
</tr>
<tr>
<td>MuxA (2)</td>
<td>Selects the data to be passed as argument A to the ALU 00 RegA, 01 PC, 10 “0”, 11 N/A</td>
</tr>
<tr>
<td><strong>MuxB(2)</strong></td>
<td>Selects the data to be passed as argumentB to the ALU</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>00</td>
<td>RegB</td>
</tr>
<tr>
<td>01</td>
<td>The sign/zero extended immediate value</td>
</tr>
<tr>
<td>10</td>
<td>N/A</td>
</tr>
<tr>
<td>11</td>
<td>“0”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ShiftA(2)</strong></th>
<th>Determines how argumentA should be shifted</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>no shifting</td>
</tr>
<tr>
<td>01</td>
<td>shift left by the amount(+1/-1) specified by RegisterB</td>
</tr>
<tr>
<td>10</td>
<td>shift left by the amount(+1/-1) specified by the immediate field</td>
</tr>
<tr>
<td>11</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ZSExtendImmed(1)</strong></th>
<th>How the 8-bit immediate field should be extended to 16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero-Extend</td>
</tr>
<tr>
<td>1</td>
<td>Sign-Extend</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ResSel(2)</strong></th>
<th>Selects which result should be sent out as the result of the OP stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>From the Concater</td>
</tr>
<tr>
<td>01</td>
<td>From the Flag Decoder</td>
</tr>
<tr>
<td>10</td>
<td>From the ALU</td>
</tr>
<tr>
<td>11</td>
<td>From the MemoryValue</td>
</tr>
</tbody>
</table>

**More details when we describe the OP stage**

<table>
<thead>
<tr>
<th><strong>Result(16)</strong></th>
<th>The result of the OP stage</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>ReadMemAddr(16)</strong></th>
<th>The address of the memory we wish to read from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MemVal(16)</strong></td>
<td>The resulting value from the read</td>
</tr>
<tr>
<td><strong>WrtMemAddr(16)</strong></td>
<td>The address of the memory we wish to write to</td>
</tr>
<tr>
<td><strong>MemWE(1)</strong></td>
<td>The WriteEnable signal for the memory</td>
</tr>
<tr>
<td><strong>MemWData(16)</strong></td>
<td>The data to be written to memory</td>
</tr>
<tr>
<td>PrevFlags(5)</td>
<td>The flags (CLFZN) from the previous cycle.</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Flags(5)</td>
<td>The new flags freshly calculated this cycle</td>
</tr>
<tr>
<td>FlagWrtMask(5)</td>
<td>Indicates whether each of the five flags should be updated i.e. a value “11011” means the C, Z and N flags should be updated</td>
</tr>
<tr>
<td>ConditionCode(4)</td>
<td>The condition code from the instruction (in case we are doing a conditional branch, set or jump)</td>
</tr>
<tr>
<td>FlagResultBit(1)</td>
<td>Whether the condition code has been satisfied by the current flags</td>
</tr>
<tr>
<td>Branch(1)</td>
<td>A signal indicating whether the current instruction is a potential branch (branch condition, jump condition or jump and link)</td>
</tr>
</tbody>
</table>

**Clock Frequency**

Our processor uses two clock frequencies - 50 MHz and 5 MHz. The on-board oscillator is set to 50 MHz, and we use a clock divider to divide it down to 5 MHz. Our processor is designed to execute one instruction per clock cycle. The 5 MHz clock is the clock that runs as fast as our pipeline can run. For example if we used SDRAM as our program memory rather than EPROM then because SDRAM has slower access times we would change the 5 MHz clock to 4.16 MHz because that is how long it would take for our longest pipeline stage to complete. The fast clock is guaranteed to always remain 50 MHz and is intended for components that either depend on a fixed clock, for example the NES Controller input and the VGA Output must both run at 60 Hz no matter what the speed of the pipeline is. There is also cases where multiple things must be done in a pipeline stage and we expose the 50 MHz clock to the stages that need this. For example the Fetch stage can only fetch 8 bits at a time from EEPROM and so it needs the 50 MHz clock so it can do two fetches per 5 MHz clock to get all of the data it needs.

The components that need the faster 50 MHz clock are

- The BlockSRAM Controller
- The FlashEPROM Controller
- The I/O Controller
- The VGA Controller
- The NES Gamepad Controller
- The Boot Sequence Controller
**ALU Design**

Our ALU is 16-bit wide, and can do addition, logical AND, OR and XOR. We can also do subtraction by using the signal ‘isNegate’, which controls whether the 2nd operand should be negated before performing the addition.

As illustrated below, the ALU is implemented as a VHDL code. We specify the behaviors and let the synthesizer build adders/subtracters/comparators for us.

The ALU computes all the flags (CLFZN) regardless of the current instruction. It is up to the FlagRegister and the FlagWriteMask to determine which flags should be updated, and which one’s ignored.
Op Stage Design

The OP stage consists of several components.

- **Shifter**: Operates on the Register A argument. It can either pass the value through, or do logical left/right shifts based on the value in the immediate mode or in RegB.

- **Extender**: Sign/Zero extend the 8-bit immediate value to 16-bit

- **Two MUX’es**: Choose what arguments are to be sent to the ALU/Concat. Candidates include RegA, RegB, extended immediate value, PC, or just zero.

- **ALU**: performs arithmetic and logical operations, i.e. ADD, AND, OR, XOR. Subtraction can also be done by negating the 2nd operand and doing an addition.
- Concater: Takes 8 bits from operandA and another 8 bits from operandB and concat them together. This is necessary to support the LUI (load upper immediate) instruction.

- FlagDecoder: Takes the condition code and flags values and determine whether the condition is satisfied. This is used for conditional branch/jump/set instructions.

- Result MUX: Chooses among the outputs of all the different components and send it as the output of the entire OP stage. The MUX control signal depends on the current instruction. i.e. If we are doing a LUI, it will choose the output from the concater. On the other hand, if we are doing an ADD, it will choose the output from the ALU.

**Register File Design**

Note that our design allows 3 simultaneous accesses to the registers- two asynchronous reads and one synchronous write.

![Register File Diagram](image-url)
Memory Interface Design

We have two completely independent memory units.

- Instruction Memory → uses the FlashEPROM
- Data Memory → uses 6x BlockSRAM

Instruction Memory

The FlashEPROM is only 8-bit wide, so we have a small state machine to control it. We read the lower byte, then the upper byte and concatenate them together.
**Data Memory**

We use 6 BlockSRAM, each configured to contain 256 Words, each 16-bit wide. This gives us a total of 1536 words. Given an 11-bit address, we use the top three bits to select the BlockRAM that will handle this address.
**Decoder Design**

The decoder is a combinational circuit. We created it as a gigantic doubly-nested case statement and let the synthesizer figure out the corresponding circuit.

Note that all signals are assigned values, regardless of whether they apply or not. i.e. we always assign the lower 8 bits to the immediate field, even if the instruction is not an op-immediate. The value will simply not be used by the OP stage.
```plaintext
case ( instruction(15 downto 12) ) is
  when "0000" =>
    case ( instruction(7 downto 4) ) is
    when ADD_EXT =>
      -- send Add signals
      branch <= '0';
      regWE <= '1';
      PC_ResSel <= '1';
      memWE <= '0';
      flagWrtMask <= "10100*";
      conditionCode <= "---";
      isNegate <= '0';
      ALU_Cp <= "00";
      resSel <= "10";
      muxA <= "00";
      muxB <= "00";
      shiftA <= "00";
      ZSextendImmed <= 4;
      when SUB_EXT =>
      -- send Sub signals
      branch <= '0';
      regWE <= '1';
      PC_ResSel <= '1';
      memWE <= '0';
      flagWrtMask <= "10100*";
      conditionCode <= "---";
      isNegate <= '1';
      ALU_Cp <= "00";
      resSel <= "10";
      muxA <= "00";
      muxB <= "00";
      shiftA <= "00";
      ZSextendImmed <= 4;
  when "0100" =>
    case ( instruction(7 downto 4) ) is
    when LOAD_EXT =>
      -- send LOAD signals
      branch <= '0';
      regWE <= '1';
```

### Instruction Decode Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Read</th>
<th>Op</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read</strong></td>
<td><strong>Op</strong></td>
<td><strong>Write</strong></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Sel A</td>
<td>Reg A</td>
<td>Sel B</td>
</tr>
<tr>
<td>ADD 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>ADDI 11-8</td>
<td>--</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>SUB 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>SUBI 11-8</td>
<td>--</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>CMP 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>CMPI 11-8</td>
<td>--</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>AND 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>ANDI 11-8</td>
<td>--</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>OR 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>ORI 11-8</td>
<td>--</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>XOR 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>XORI 11-8</td>
<td>--</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>MOV -- 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>MOVI --</td>
<td>--</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>LSH 11-8 3-0</td>
<td>--</td>
<td>01</td>
<td>--</td>
</tr>
<tr>
<td>LSHI 11-8</td>
<td>--</td>
<td>10</td>
<td>7-0</td>
</tr>
<tr>
<td>LUI 11-8</td>
<td>--</td>
<td>00</td>
<td>7-0</td>
</tr>
<tr>
<td>LOAD -- 3-0</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>STOR 11-8 3-0</td>
<td>--</td>
<td>00</td>
<td>--</td>
</tr>
<tr>
<td>Scond --</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Bcond --</td>
<td>--</td>
<td>1</td>
<td>7-0</td>
</tr>
<tr>
<td>Jcond --</td>
<td>3-0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>JAL -- 3-0</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

***Note*** A-B means all bits in the inclusive range [A,B]. i.e. 7-0 is an 8-bit value taken from bit 7,6,5,4,3,2,1,0 of the instruction.
I/O Interface Design

We use the top 5 bits of the address to indicate which device is being addressed. At cycle1 (20ns later) we pass the WriteEnable to the selected device. We also use the top 5 bits to determine which device’s return value should be passed back to the processor.

Note that we consider the Data Memory (BlockSRAM) as another I/O device for simplicity.
**VGA Controller**

Our VGA controller is double buffered. We use two BlockSRAM units. One for each video buffer. Each buffer is configured to be 1024 words x 2 bits/word. This gives us a theoretical resolution of 64x32 pixels with 4 colors. However, we throw away the last 2 rows to make the hardware simpler. So our real resolution is 64x30 x 4 colors. The palette is hard-coded into the controller.

At any given time, we have a “back” and a “front” buffer. The front buffer is used to display to the monitor. The VGA controller asks for the color of every pixel, starting from the top-left corner to the lower right, and relays this information to the monitor. In the mean time, the back buffer is being written to by the running program through the I/O controller.

When the entire front buffer has been sent to the monitor, the VGA controller performs a vertical sync, as part of the VGA protocol. It sets the signal VsyncStarting, which can be observed by the processor. If the processor chooses to, it can send a flipPage signal, causing the VGA controller to swap the front and back buffer.
Flipping the pages is done by simply invert the signal that controls all the multiplexers. If a buffer was receiving the address from the I/O controller, switch it so it received from the VGA controller and vice versa. Also, we toggle the 2 WriteEnable signals.

The VGA Controller itself is implemented as three state machines. To get the timing of all signals to conform to the VGA protocol, we use a clock divider to obtain a 3.125MHz clock (divide 50MHz by 16).

StateMachine #1 99 states. Control the sending of one row of pixels

StateMachine #2 480 states. Controls the sending of 30 rows of pixels. Note that the state machine changes state when the first machine wraps around (from state 98 to 0).
StateMachine #3 52371 states. Controls the VerticalSync, sending pixels of the entire page and page flipping. Also, when this state machine wraps around (from state 52370 to 0), we reset the 2nd machine back to state zero as well, so it starts at the first row again.

All three state machines are implemented as counters that count up and wrap around when they reach their respective last state.

We put together the address of the pixel to be addressed by taking pieces from the 1st and 2nd state machines’ counters.

Finally, we enforce all BlankPeriods by setting all components of the RGB color to zero during the blank time.
To write pixels to the VGA buffer, send the color (between 0 to 3) to any address between 0xE800 (top-left corner) to 0xEF80 (lower-right corner).

To flip the video buffer, keep reading from address 0xF800 until the value is non-zero. Write the value zero to address 0xF000, wait 16 cycles, and write the value one to the same address again.

**Sound Device**

Our goal with sound output was to be able to emit simple tones. To achieve this we built an analog sound circuit that we interface with our processor. Our sound circuit takes four digital signals. One enable input and three inputs that control the frequency of the emitted tone.

Our inspiration for this circuit was taken from a standard electronic kit manual. We choose this circuit over other designs because it gives us decaying resonance when changing tones rather than an immediate tone change. These decaying tones seemed more appropriate for the Space Invaders game than tones that cut off immediately. The schematic of our circuit follows.

![Schematic diagram of the sound circuit](image-url)
The sound device is controlled by 4 one-bit signals - enable, bit2, bit1 and bit0. To control these signals, write to address 0xD800. The lower 4 bits will be used as the enable, bit2, bit1 and 0 respectively. The value stays there until it is overwritten.

**NES Game pad Controller**

The idea behind the NES Game pad controller is to get a standard console input. There was some discussion on whether we should take keyboard input or game pad input. However being that we are designing a game console system a standard game console controller input seemed more appropriate. For details of what pins to map to what look in nes_controller.vhd.

The NES controller is a state machine that completes a cycle every 60 Hz. A diagram of the state machine is shown below. During the latch stage a reset signal is sent to the NES Controller which causes the NES Controller to read what buttons store that information. The “A First” stage allows the NES Controller time to start sending the value of the A button on the data line. The value of the A button is then read in during the “A Mid” stage and the reset signal is set low. Then during the “B First” stage the next signal is set hi. This causes the NES Controller to send out the value of the B button on the data line. This value is read in during the “B Mid” stage and the next signal is set to low. This pattern continues as shown in the stage diagram below until the values of all of the buttons are read in. During the extra stage the output values of the NES Interface Controller are updated. And then the process is repeated.

The NES Paddle’s buttons are active low so the data line is ‘1’ when the button is not being pressed and ‘0’ when the button is being pressed. So the programmer can have a more intuitive interface were the value of ‘1’ symbolizes the button being pressed down these values are reversed.
Tool Set

VHDL Compiler & Simulator : Symphony EDA
- The command line tools are available for free at http://www.symphonyeda.com/
- very fast compile & simulation time. Very low overhead.
- usage: vhdlp [filename] to compile
  vhdle [filename] to simulate
- very intuitive to use
- generates meaningful error messages. We even wrote our own translator so
  the VisualStudio.NET and emacs can parse the error message and go to the offending line
  of code immediately.
- allows customization by specifying additional flags
- can generate signal timing diagram

Synthesis and Place & Route : Xilinx WebPack
- allows the entire system to be rebuilt with one click
- reasonably easy to use
- generates meaningful reports of how the code is translated into hardware

CVS Version Control System
- We set CVS on our own computer that is permanently online.

CASM Compiler/Assembler
- This is a compiler/assembler that we created on top of a standard C++ compiler. See
  the programming manual for further details.
- Can take an assembly code, compile it, generate a VHDL testbench and run the code in
  the simulator and automatically determines if the code pass or fail

toXes
- This is a small tool we created that can turn binary code generated by our CASM compiler into .XES file, which is used to download code onto the actual hardware.

**Testing Framework**

Proper testing was very important throughout the project. As we developed VHDL components we have a test framework that allows us to type “make test” which would run VHDL testbenches. Every VHDL file had a separate testbench. Someone other than the person who wrote the VHDL implementation wrote the testbench. Our design is about 9,000 lines of VHDL code and 10,000 lines of VHDL testing code, and 2,400 lines of CASM pipeline testing code.

The following is a screenshot of running our VHDL testbench:
After our processor was fully put together we wrote an extensive serious of CASM test programs. We built a testing framework that compiles the test programs down and runs them in simulation on the processor. Once again “make test” runs this testing framework. The following is a screenshot:
After we got our processor to boot on hardware we ran each of the CASM test programs on hardware and they returned the correct result first try and so we did not make an automated test program for hardware.
Programming Manual

Our approach to compiling code down to our processor is unique. We wrote C++ functions that when called emit assembly code. So we have an “void add(int immediate, cr16_int register)” function that when invoked checks the values passed into it to ensure that they are valid and emits the correct assembly code. We call it CASM with the ‘C’ representing the C programming language and ASM meaning assembler. This approach is non-standard, however it is very appropriate for our situation. CASM has all of the functionality that we desire including, automatic hazard correction, and the ability to branch to labels rather than just numeric offsets. CASM was developed by Usit and Russ in ten man hours. It would not have been possible to get such a fully functional assembler in such a short amount of time had we taken the conventional approach.

After we developed a working assembler Usit came up with the idea of further extending CASM with if statements, while loops, expression evaluation, and inline functions. Russ further extended the design by adding for loops, function calls, and developed a suggested programming style. Extending the CASM Assembler into the CASM Programming language took twenty man hours. Had we taken a conventional approach it would have been impossible to develop such a full featured compiler in such a short amount of time.

With the CASM Language is appropriate for larger programs. Usit has developed a Space Invaders game that is approximately 1,600 lines of CASM Language code that translates to about 10,000 lines of assembly code. Russ’s Tetris game is approximately 2,100 lines of CASM that translates into about 20,000 lines of assembly code. Because of how the CASM Language is structured about half of the assembly code is nop instructions that have been inserted by the compiler’s hazard detection unit. Most of these nops() could be avoided by programming in strait CASM Assembly. However this is a fair trade off because both Tetris and Space Invaders are performant games and it would not have been possible to develop them in our time constraints had we programmed in strait assembler.

The Tetris program source code has been commented to help the first time CASM programmer understand what is going on. We recommend that anyone interested in
developing a substantial CASM program refer to the Tetris source code for examples of how to use the features of the language. And for a suggested programming style.

When programming CASM there are some points that to keep in mind.

- If doing manual branching, meaning using the bCond or jCond functions it is recommended that you always have a flushPipeline function call immediately following the branch instruction. The branch actually does not take place until 4 instructions after the branch instruction because it takes that long to get through the pipeline. The flushPipeline function is the same thing as just putting in four nops so the processor does nothing useful until after the decision to branch or not is decided. If for performance reasons you would like to place instructions in these four slots then please keep the following in mind. The code in the four slots after the branch instruction is always executed whether the branch is taken or not. The hazard detection unit only looks for hazards in a linear way. Meaning if you set a register to a value in the last slot before the jump and then a jump occurs the register will not get it’s updated value until four instructions after the jump. Where as in linear program execution the hazard detection unit will insert nops on an as needed bases so the programmer can assume that a register automatically takes a new value immediately after being set. Another issue that the programmer must pay attention to if desiring to use the 4 slots after the branch is to ensure that the hazard detection unit does not detect a hazard in those slots and fill in nops. Then the branch will occur before executing the intended code. Another issue to be aware of if using these four slots is it is undefined behavior of our hardware if a load instruction is in the read stage of the pipeline while the store instruction is in the write stage of our pipeline. Under all cases except for doing manual branching without using the flushPipeline function CASM detects this and inserts a nop instruction as appropriate. Only when using the four slots after branch must the programmer be aware of this. While CASM has the functionality to allow you to use the slots after the branch in performance critical situations, the programmer is strongly urged to never do this. You should always have a flushPipeline() function after a manual branch instruction. And this is only in effect for CASM
Assembly. The CASM Programming language functions to do loops and if statements automatically does flushPipeline for you.

- Registers 10, 11, 12, and 13 are reserved for the CASM Programming Language for doing expression parsing, conditional codes and loops.
- CR16 Programming convention is to use Register 14 for the return address from a function and to use Register 15 as the stack pointer.
- Save yourself a lot of time and use the helper functions found in the Tetris and Space Invaders programs. Usit wrote memInit(), memLoad, and memSave(). Russ wrote a cr16 for loop some mult and div functions, and code to do non-inline function calls.
- Even though we require gcc 2.95.3 running in the cygwin environment to compile the CASM programs. Other versions of gcc may work as well but are untested. Even though we require gcc it is strongly recommended that you do all code development in Microsoft Visual Studio .NET. The Visual Studio editor can give you much needed context sensitive help on the arguments of each function when you mouse over the function. These programs would not have been possible had we used a text editor like emacs to develop them.
- When programming for this system remember that it is something that we developed for our own use. It is not something that we have prepared for commercial use. While we have offered this tool to the rest of the class as of the time of this writing we are not aware of CASM being used outside of our internal development. Therefore the same people that implemented the language are the only people who have used it. We understand the code of the CASM Compiler and we understand how everything drops down. If you want to use CASM it is strongly suggested that you read the code of the CASM Compiler and read and understand the code of the Tetris program. Because of our design no amount of documentation can replace the importance of just reading our source. If you try for the fast track and just develop for our system without doing what we have urged you program at your own risk.
Operation Manual

- Make sure that the VGA cable, power, LPT cable, NES gamepad and sound device are all connected.
- The processor should emit static frequency noise until it is programmed.
- Load pipeline_synth.bit into the FPGA and the code to execute (i.e. invader.xes or tetris.xes) into the FlashEPROM.
- The Noise should go away when the loading finishes.
- The processor will automatically boot and execute the code.
- If you want to re-load the Flash, you must first disable the processor. This can be done in two ways. Either unplug and replug it, or press the push button on the main board (near the PS2 port).

Simulation Testing

We use VHDL testbench programs to test our circuits. The advantage of using testbenches is we get more organization such as conditional instructions, loops and procedure calling. All test cases can be invoked from the command prompt. We create Makesfiles so that all available test cases can be executed by typing a single command. This proves very useful when we want to make modifications to the circuit. It allows us to follow the eXtremeProgramming approach.

Xess Board Testing

Test cases used in the simulation are manually loaded on to the board and execute one at a time. This was somewhat a tedious task, but because we only need to do it once (after everything passes in simulation), we decided not to automate the process.

Initially, we either wire up LEDs to the board or use a digital oscilloscope to determine whether it works. We export the lower bits of register zero to some of the free pins. Later on, after we have the VGA controller working, we switched to using the monitor instead.

Assembly Test Program
We add functionalities to the CASM tool so it can not only compile the code, but also spawn processes to generate a VHDL testbench out of the code, invoke the VHDL compiler and simulator, execute the code in simulation and determine whether the test case fails. To do this, the programmer needs to have a special command in the assembly code indicating what is the expected result. The simulator will check the simulated result against this and complain if they do not match.

Example of the First Few Lines of a Test Code

```assembly
activateHazardDetectionUnit();
compileAndSimulate(__FILE__); // Tell the compiler to simulate current file
expectResult("0000000011111000"); // Tell simulator the expected result of reg0
runFor(40000); // Number of cycles the simulator will run?
```

```
// Actual Testing Code
// ret r14
cr16_int arg1(r1);
cr16_int res(r0);
sp = 0;
arg1 = 11;
res = 0;

.............
```

We wrote code to test all individual instructions, and also more complex programs such as summing 1+2+3+…100, calculating Fibonacci numbers by both the recursive and iterative methods. We also write large-scale games such as Tetris and Space Invaders (discussed below).
The CASM Compiler/Assembler

As illustrated in the Programming Manual our CASM language is very robust for a home-brew assembler. That is because we adopt a non-standard method and take advantage of existing C++ compiler.

Rather than writing a complete assembler that parses an assembly code and generates the corresponding binary code, we use g++ to parse our CASM code. The CASM code must be written in C++-style syntax, which is the reason why commands like movi and add need parenthesesizes and semicolons. We created the CASM library,
which is simply a C++ code implementing functions like *movi* and *add*. To compile the CASM code, the user uses g++ which generates a UNIX executable. The user must then execute the resulting executable which will generate the binary code executable on our CR16 machine.

Building on top of an existing compiler has many advantages. First, we never need to parse or check the syntax of any code. We can concentrate on adding functionality right from the beginning. We also get features like macros, constant declaration, variable declaration, enumeration, function calls and OOP for free. This free us up to spend more time adding high-level features to our “assembler” such as variable-register mapping, expression parsing, conditional code and while-loop.

Expression Parsing

We support Expression Parsing by overloading all basic operators (=, +, -, ++, <=, != …) so that when two expressions are operated together, a new expression node is created and returned. Also, we provide constructors that can turn immediate values and registers into expression nodes.

```cpp
class Expr {
    Expr( int immed ) { ..... }
    Expr( Reg r ) { ..... }
    ....................
}

Expr operator+ ( Expr l, Expr r ) { return Expr( OP_ADD, 0, new Expr(l), new Expr(r)); }
Expr operator- ( Expr l, Expr r ) { return Expr( OP_SUB, 0, new Expr(l), new Expr(r)); }
.......
To parse the resulting expression tree, we start from the root and traverse in post order. We reserve registers 10, 11, 12 and 13 as temporary storages.
How if-then-else is Implemented

Generate 2 system labels.  
=> L30, L31

```
cr16_if( x+1<y-z );
    _some_code_1_
    cr16_else();:
    _some_code_2_
    cr16 endif();
```

If( !(x+1<y-z) ) GotoLabel(L30)  
_Label(L30)_  
_Label(L31)_  
_Thunk_  
_Goto(L30)_  
_Thunk_  
_Goto(L31)_

How while-endwhile is Implemented

Generate 2 system labels.  
=> L24, L25

```
cr16_while( x+1<20 );:
    _some_code_
    cr16 endwhile();
```

```
NodeType = "endwhile"
condition = "x+1<20"
labelToInsert = "L24"
labelToGoBack = "L25"

NewLabel(L24)  
NewLabel(L25)
```

_Goto(L24)_  
_Thunk_  
_Goto(L25)_
Show-Off Applications

Tetris

Tetris was written to show that our processor is capable of running games that are still popular today. Tetris was written first to work and then it was expected to do some manual speed tuning to make the game fast enough. However the speed tuning stage was never needed because our system was fast enough. Tetris is over 2,000 lines of CASM code it compiles down to about 10,000 assembly instructions. As the player destroys more lines the game becomes faster until it becomes to fast to play. At first the game emits a ticking noise that becomes more stressful as the game speed increases. To hear it in action view the playing_tetris.mpg included on the CD enclosed with this final report. There is also a sound made when you destroy a line.

Space Invaders

Space Invaders is another of our demo program. It is reasonably large- 1,418 lines of CASM code, or 10,411 assembly instructions. The program is designed to show
what our CR16 is capable. It uses multiple sprites for the player’s ship and two types of aliens. There is parallax scrolling starfield, basic particle explosion effects, intro screen, score counter, etc. The program performs many complicated operations. When flattened out, Space Invaders has about 7 levels of nesting code (if-then-else and while-loop) and many nested expressions. For full details, please read the source code invaders.cpp

**Conclusions**

The course has been immensely fun. We would like to thank Professor Brunvand for giving us the flexibility to do what we did. We would not have enjoyed as much if we were told to stick with the standard tools and create a plain, regular processor. Quite surprisingly, there was not one thing that went wrong. We moved our own self-appoint deadlines several times, always moving them closer after tasks that were predicted to be difficult were finished relatively quickly. Since the beginning, we had some worries that our perfectly simulated VHDL code might not translate nicely into working hardware. Our fear were proven unfounded.
One point we would really like to make is that the class should be advised to change the tool set. Some of us who took CS3700 using PowerView still remember with agony how it was like. The tool was in many ways unpredictable, silently breaking down without an understandable error message. It also has a very long development cycle. If a change is made in one of the VHDL file, it has to be recompiled (with one menu), resimulated (close that menu, load the simulator), resynthesized (by a different tool), reexported, reimported ...and so on. This made it very difficult to try different approaches or fix bugs. The tools we chose for ourselves directly address these issues.

We use a command line compiler/simulator that take only a few seconds to load and execute. It was quite robust that we were able to develop our own tools to suit exactly our needs. We create scripts and programs so that assembly files can be compiled, added to the test database and run all tests on all hardware components, with a single command. We can make modification to some of the files, rebuild the entire system and run all the test cases with a single command, and quickly see how the change effects the entire system. The Xilinx WebPack, another software we use, is also much more friendly to work with than PowerView. It has a one-click-to-rebuild-them-all button, can generate useful reports of how the synthesis and place and route processes, and in general, reasonably predictable. We also create our own assembler/compiler, which we think are much more powerful and flexible than the regular assembler. Following a non-standard convention, we build our assembler on top of a standard C++ compiler, which allows us to do very high level coding such as structured code, function calls, enumeration, OOP etc. We have written complex programs like SpaceInvaders and Tetris, which involve over 10,000 assembly instructions. We simply could not have done it with a regular assembler.

We did not learn much new theories in this course. Instead, we get the chance to apply knowledge we learned in CS3700 and CS3810 into real, functioning machines. We particularly enjoy creating our own system requirements, making trade-off decisions, designing our own architecture and make it work. We feel we understand computer architecture much better now that we have experienced it first hand. It is much more fun and also more interesting to build just one rather than learn about ten of them. Overall,
we think this course is very beneficial, and we are proud to have taken it, and accomplished what we did.
Appendixes

VHDL Source Code

NOTE! Our processor is 9,000 lines of VHDL code. If we were to print all of it we would extend the size of this report by 108 pages. It is highly unlikely that anyone would ever read all of that and if they did they would probably wish the whole time that they could just look at the code on the computer with a text editor and grep. Therefore we have only printed a small portion of our VHDL design and have enclosed a CD that contains all of VHDL code.

ALU_16.vhd

-- Ben Holt bholt@eng.utah.edu
-- Justin Olson justino@cs.utah.edu
-- Russell Christensen rchriste@cs.utah.edu
-- Usit Duongsaaduongsaa@cs.utah.edu
-- CS3710

-- Component : alu_16
-- Person Responsible : Justin Olson
-- ALU 16 Bits

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity alu_16 is

port ( op1 : in std_logic_vector(15 downto 0); -- 1st operand
op2 : in std_logic_vector(15 downto 0); -- 2nd operand
isNegate : in std_logic; -- Need to negate for subtraction
aluOp : in std_logic_vector(1 downto 0); -- Op code for operation
result : out std_logic_vector(15 downto 0);

-- flags
constant ADD_OPC : std_logic_vector := "00";
constant AND_OPC : std_logic_vector := "01";
constant OR_OPC : std_logic_vector := "10";
constant XOR_OPC : std_logic_vector := "11";
end alu_16;

architecture alu_16_arch of alu_16 is

-- # Define all internal signals
signal dataAdd : signed(15 downto 0);
signal dataAnd : std_logic_vector(15 downto 0);
signal dataOr : std_logic_vector(15 downto 0);
signal dataXor : std_logic_vector(15 downto 0);
signal op1_signed,op2_signed : signed(15 downto 0);
signal op2eff : signed(15 downto 0);
signal op2eff_vec : std_logic_vector(15 downto 0);

begin -- alu_16_arch
  op1_signed <= signed(op1);
  op2_signed <= signed(op2);

  process(ALUop,dataAdd,dataAnd,dataOr,dataXor,op1,op2,isNegate,op1_signed,op2_signed)
    variable lFlag : std_logic; -- # Used as internal variable
  begin
    case ALUop is
      when ADD_OPC => result <= std_logic_vector(dataAdd);
      when AND_OPC => result <= dataAnd;
      when OR_OPC => result <= dataOr;
      when XOR_OPC => result <= dataXor;
      when others => null;
    end case;

    -- Status Flags
    -- L : if result of subtraction is negative
    clfzn_flags(3) <= dataAdd(15);

    -- F : if the sign of the result is different from what you expect
    if isNegate='0' then
      if (op1(15)="0" and op2(15)="0" and dataAdd(15)="1") or
         (op1(15)="1" and op2(15)="1" and dataAdd(15)="0") then
        clfzn_flags(2) <= '1';
      else
        clfzn_flags(2) <= '0';
      end if;
    else
      if (op1(15)="0" and op2(15)="1" and dataAdd(15)="1") or
         (op1(15)="1" and op2(15)="0" and dataAdd(15)="0") then
        clfzn_flags(2) <= '1';
      else
        clfzn_flags(2) <= '0';
      end if;
    end if;

  end process;

end architecture;
end if;
end if;

-- Z : if both ops are identical
if opp1_signed = opp2_signed then
  clfznflags(1) <= '1';
else
  clfznflags(1) <= '0';
end if;

-- N : first look at signs of operand, if they're different
-- we can tell right away which one is greater
-- otherwise subtract them and look at sign of result
if opp1(15)='1' and opp2(15)='0' then
  clfznflags(0) <= '1';
elsif opp1(15)='0' and opp2(15)='1' then
  clfznflags(0) <= '0';
else
  clfznflags(0) <= dataAdd(15);
end if;

-- C
if ( opp1(15)='0' and opp2_vec(15)='1' and dataAdd(15)='0' ) or
 ( opp1(15)='1' and opp2_vec(15)='0' and dataAdd(15)='0' ) or
 ( opp1(15)='1' and opp2_vec(15)='1' ) then
  clfznflags(4) <= '1';
else
  clfznflags(4) <= '0';
end if;
end process;

--# Create all possible permutations of Data that we might need
--# and then select the correct one at the end
process(opp1_signed, opp2_signed, isNegate, opp1, opp2, opp2eff)
begin
  if isNegate = '0' then
    opp2eff <= opp2_signed;
  else
    opp2eff <= -opp2_signed;
  end if;

  dataAdd <= opp1_signed + opp2eff;
  opp2eff_vec <= std_logic_vector(opp2eff);

  dataAnd <= opp1 and opp2;
  dataOr  <= opp1 or opp2;
  dataXor <= opp1 xor opp2;
end process;

end alu_16_arch;
pipeline_synth.vhd

-- Ben Holt bholt@eng.utah.edu
-- Justin Olson justino@cs.utah.edu
-- Russell Christensen rchriste@cs.utah.edu
-- Usit Duongsaad duongsaad@cs.utah.edu
-- CS3710

-----*** SYNTHESIZABLE. NOT SIMULATABLE ***

-- Component : pipeline
-- Person Responsible : Usit
-- What does it do? The entire pipeline of our computer

-- 'reset' will reset PC to zero
-- if 'enable' is '0', the entire pipeline freezes and maintain current
state
-- has 2 input clock signals. clk10x must be 10x faster than clk

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity pipeline_synth is

port ( 
  isChipReady_p83 : in std_logic;

  --both set_reset and set_enable are one-way. Once you set them to
  '1' they
  --stay that way until you clear them and set clear_both_p93
  set_reset : in std_logic;
  set_enable : in std_logic;
  clk10x : in std_logic;
  flash_data_out : in std_logic_vector(7 downto 0);

  -- these signals are here so we can test (check output)
  flash_ce_41 : out std_logic;
  flash_ce_43 : out std_logic;
  flash_a17_56 : out std_logic;
  --flash_a16_63 : out std_logic;
  flash_we_58 : out std_logic;
  flash_reset_59 : out std_logic;
  flash_mem_addr_buff : out std_logic_vector(16 downto 0);

  vga_Hsync_p23 : out std_logic;
  vga_Vsync_p26 : out std_logic;
  vga_red1_p13, vga_red0_p12 : out std_logic;
  vga_gre1_p20, vga_gre0_p19 : out std_logic;
  vga_blu1_p22, vga_blu0_p21 : out std_logic;

  nes_data_p80 : in std_logic;
  nes_latch_p78 : out std_logic;
  nes_pulse_p94 : out std_logic;
);
sound_bits_p87_to_p84 : out std_logic_vector(87 downto 84);

-- clear_both set the reset & enable back to 0
-- set_reset and set_enable set the signal to 1
-- We do this to prevent switch bouncing
-- this way, a switch can set a signal one way but not back

clear_both_p93 : in std_logic
);
end pipeline_synth;

architecture pipeline_synth_arch of pipeline_synth is

--------------------------------------------------------------------------

component stage_op

port (condCode : in std_logic_vector( 3 downto 0);
prevFlags : in std_logic_vector(4 downto 0);
regA : in std_logic_vector(15 downto 0);
regB : in std_logic_vector(15 downto 0);
ZSextendImmed : in std_logic;
immed : in std_logic_vector( 7 downto 0);
PC : in std_logic_vector(15 downto 0);
shiftA : in std_logic_vector( 1 downto 0);
memVal : in std_logic_vector(15 downto 0);
resSel : in std_logic_vector( 1 downto 0);
muxB : in std_logic_vector( 1 downto 0);
muxA : in std_logic_vector( 1 downto 0);
aluOp : in std_logic_vector( 1 downto 0);
isNegate : in std_logic;
outFlags : out std_logic_vector( 4 downto 0);
outResult : out std_logic_vector(15 downto 0);
outFlagResultBit : out std_logic)
);
end component;

--------------------------------------------------------------------------

component stage_decode

port (instruction : in std_logic_vector(15 downto 0);
branch : out std_logic;
regWE : out std_logic;
PC_ResSel : out std_logic;
memWE : out std_logic;
flagWrtMask : out std_logic_vector(4 downto 0);
regWrtSel : out std_logic_vector(3 downto 0);
conditionCode : out std_logic_vector(3 downto 0);
isNegate : out std_logic;
ALU_Op : out std_logic_vector(1 downto 0);
resSel : out std_logic_vector(1 downto 0);
muxA : out std_logic_vector(1 downto 0);
muxB : out std_logic_vector(1 downto 0);
immediate : out std_logic_vector(7 downto 0);
shiftA : out std_logic_vector(1 downto 0);
ZSextendImmed : out std_logic;

51
component bit1_reg
port (  
clock, enable : in std_logic;
indata : in std_logic;
outData : out std_logic );
end component;

component bit2_reg
port (  
clock, enable : in std_logic;
indata : in std_logic_vector(1 downto 0);
outData : out std_logic_vector(1 downto 0) );
end component;

component bit4_reg
port (  
clock, enable : in std_logic;
indata : in std_logic_vector(3 downto 0);
outData : out std_logic_vector(3 downto 0) );
end component;

component bit5_reg
port (  
clock, enable : in std_logic;
indata : in std_logic_vector(4 downto 0);
outData : out std_logic_vector(4 downto 0) );
end component;

component bit8_reg
port (  
clock, enable : in std_logic;
indata : in std_logic_vector(7 downto 0);
outData : out std_logic_vector(7 downto 0) );
end component;

component bit16_reg
port (  
clock, enable : in std_logic;
indata : in std_logic_vector(15 downto 0);
outData : out std_logic_vector(15 downto 0) );
end component;

component mem_flash_sim
port (  
wrData : in std_logic_vector(15 downto 0);
wrt : in std_logic;

addr : in std_logic_vector(13 downto 0);
clk : in std_logic;
cout : out std_logic_vector(15 downto 0);
);
end component;

component mem_flash
port (    
addr : in std_logic_vector(15 downto 0);
clk : in std_logic;
cout : out std_logic_vector(15 downto 0);
flash_data_out : in std_logic_vector(7 downto 0);
flash_mem_addr : out std_logic_vector(16 downto 0);
reset : in std_logic
);
end component;

component mem_sram_sim
port (    
wrtData : in std_logic_vector(15 downto 0);
wrt : in std_logic;
addr : in std_logic_vector(13 downto 0);
clk : in std_logic;
cout : out std_logic_vector(15 downto 0)
);
end component;

component RegFile
port(
    selectRegA : in std_logic_vector(3 downto 0);
    selectRegB : in std_logic_vector(3 downto 0);
    selectRegWrt : in std_logic_vector(3 downto 0);
wrt : in std_logic;
clk : in std_logic;
reset : in std_logic;
wrtData : in std_logic_vector(15 downto 0);
readDataA : out std_logic_vector(15 downto 0);
readDataB : out std_logic_vector(15 downto 0);
firstData : out std_logic_vector(15 downto 0)
);
end component;

component mux2
port(
    input0 : in std_logic_vector(15 downto 0);
    input1 : in std_logic_vector(15 downto 0);
    selector : in std_logic;
cout : out std_logic_vector(15 downto 0)
);
end component;
component flag_reg
  port (
    clock : in std_logic;
    clfzn_enable : in std_logic_vector(4 downto 0);
    in_flags : in std_logic_vector(4 downto 0);
    out_flags : out std_logic_vector(4 downto 0) );
end component;

component clock_divider is
  port (
    clk      : in std_logic;
    reset    : in std_logic;
    clk_out  : out std_logic);
end component;

component io_controller is
  port (
    -- interface to the processor
    reset     : in std_logic;
    clk10x     : in std_logic;
    data_in    : in std_logic_vector(15 downto 0);
    addr       : in std_logic_vector(15 downto 0);
    wrtEn      : in std_logic;
    data_out   : out std_logic_vector(15 downto 0);
    vga_Hsync_p23 : out std_logic;
    vga_Vsync_p26 : out std_logic;
    vga_red1_p13, vga_red0_p12 : out std_logic;
    vga_grel_p20, vga_gre0_p19 : out std_logic;
    vga_blul_p22, vga_blu0_p21 : out std_logic;
    nes_data_p80 : in std_logic;
    nesLatch_p78 : out std_logic;
    nes_pulse_p94 : out std_logic;
    sound_bits_p87_to_p84 : out std_logic_vector(87 downto 84)
  );
end component;

constant constOne          : signed(15 downto 0) :=
"0000000000000001";

signal    clk        : std_logic;
signal reset, enable : std_logic;

-- signals are prefixed with sN_ where N is [ 1,8]
-- N tells which stage of the pipeline the signal is in
-- 0 = input of fetch  1 = output of fetch
-- 2 = input of decode  3 = output of decode
-- 4 = input of read    5 = output of read
-- 6 = input of op      7 = output of op
-- 8 = input of write  9 = output of write
-- i.e. s2_instruction is the instruction signal going into the Decode stage

-- separator between 2 stages are prefixed with sepMN, M,N are layer#

signal s0_pc, s1_pcPlusOne : std_logic_vector(15 downto 0);
-- note we don't need pc+1 in s3,5,7,9 because we don't modify it.
s3==s2, s5==s4 and so on
signal s2_pcPlusOne, s4_pcPlusOne, s6_pcPlusOne, s8_pcPlusOne :
std_logic_vector(15 downto 0);
signal s1_instruction, s2_instruction : std_logic_vector(15 downto 0);

-- signals from decode to read
signal s3_selRegA, s4_selRegA : std_logic_vector( 3 downto 0);
signal s3_selRegB, s4_selRegB : std_logic_vector( 3 downto 0);

-- signals from decode to op
signal s3_ZSextendImmed, s4_ZSextendImmed, s6_ZSextendImmed :
std_logic;
signal s3_shiftA, s4_shiftA, s6_shiftA :
std_logic_vector(1 downto 0);
signal s3_immed, s4_immed, s6_immed :
std_logic_vector(7 downto 0);
signal s3_muxA, s4_muxA, s6_muxA :
std_logic_vector(1 downto 0);
signal s3_muxB, s4_muxB, s6_muxB :
std_logic_vector(1 downto 0);
signal s3_resSel, s4_resSel, s6_resSel :
std_logic_vector(1 downto 0);
signal s3_ALUop, s4_ALUop, s6_ALUop :
std_logic_vector(1 downto 0);
signal s3_isNegate, s4_isNegate, s6_isNegate :
std_logic;
signal s3_condCode, s4_condCode, s6_condCode :
std_logic_vector(3 downto 0);
signal s3_flagWrtMask, s4_flagWrtMask, s6_flagWrtMask :
std_logic_vector(4 downto 0);

-- signals from decode to write
signal s3_branch, s4_branch, s6_branch, s8_branch : std_logic;
signal s3_regWE, s4_regWE, s6_regWE, s8_regWE : std_logic;
signal s3_memWE, s4_memWE, s6_memWE, s8_memWE : std_logic;
signal s3_PCResSel, s4_PCResSel, s6_PCResSel, s8_PCResSel :
std_logic;
signal s3_regWrtSel, s4_regWrtSel, s6_regWrtSel, s8_regWrtSel :
std_logic_vector(3 downto 0);

-- signals from op to write
signal s7_result, s8_result :
std_logic_vector(15 downto 0);
signal s7_flagResultBit, s8_flagResultBit :
std_logic;
signal s8_memWrtAddr : std_logic_vector(15 downto 0);

-- signals from write to fetch
signal s9_PCBranch : std_logic_vector(15 downto 0);
signal s9_PCsel : std_logic;

signal s9_regWrtData : std_logic_vector(15 downto 0);

-- op's internal
signal s6_prevFlags, s7_flags : std_logic_vector(4 downto 0);

-- register file signals
signal s5_regA, s6_regA : std_logic_vector(15 downto 0);
signal s5_regB, s6_regB : std_logic_vector(15 downto 0);

-- memory signals
-- is only a constant because we don't have memory yet.
signal s5_memVal : std_logic_vector(15 downto 0);
signal s6_memVal : std_logic_vector(15 downto 0);
signal s9_memAddr : std_logic_vector(15 downto 0);
-- signal flash_data_out : std_logic_vector(7 downto 0);
signal flash_mem_addr : std_logic_vector(16 downto 0);

constant instrMemWE : std_logic := '0';
constant instrMemWAddr : std_logic_vector(13 downto 0) := "00000000000000";
signal reg00 : std_logic_vector(15 downto 0);

constant true_bit : std_logic := '1';

-- when programming the FLASH (in testing), instrAddr is instrMemWAddr, otherwise, it's PC
signal instrAddr : std_logic_vector(15 downto 0);

signal bootCounter : integer range 0 to 9;

----------------------------------------
begin

-- debounce the control buttons
-- debounce_reset: process(clk10x, clear_both_p93, set_reset)
-- begin
-- if clk10x'event and clk10x='1' then
-- if clear_both_p93 = '0' then
-- reset <= '0';
-- else
-- if set_reset = '1' then
-- reset <= '1';
-- end if;
-- end if;
-- end if;
-- end process;
debounce_enable: process(clk, clear_both_p93, set_enable)
begin
  if clk'event and clk='1' then
    if clear_both_p93 = '0' then
      enable <= '0';
    else
      if set_enable = '1' then
        enable <= '1';
      end if;
    end if;
  end if;
end process;

bootSequence: process (clk10x, isChipReady_p83)
begin
  if isChipReady_p83 = '0' then
    reset <= '1';
    enable <= '0';
    bootCounter <= 0;
  elsif clear_both_p93 = '0' then
    bootCounter <= 9;
  else
    if clk10x'event and clk10x='1' then
      if bootCounter = 8 then
        enable <= '1';
      elsif bootCounter = 4 then
        reset <= '0';
        bootCounter <= bootCounter + 1;
      elsif bootCounter = 9 then
        reset <= '1';
        enable <= '0';
      else
        bootCounter <= bootCounter + 1;
      end if;
    end if;
  end if;
end process;

clockDiv: clock_divider port map( clk10x, reset, clk );

incPC: process( s0_pc )
begin
  variable signed_s0_pc, signed_s1_pcPlusOne : signed(15 downto 0);
  signed_s0_pc := signed(s0_pc);
  signed_s1_pcPlusOne := signed_s0_pc + constOne;
  s1_pcPlusOne <= std_logic_vector(
    signed_s1_pcPlusOne );
end process;

newPC: process( reset, clk, enable, s9_PCsel, s9_PCBranche, s1_pcPlusOne )
begin
if reset='1' then
  s0_pc <= "0000000000000000";
else
  if clk'event and clk='1' then
    if enable='1' then
      if s9_PCsel = '1' then
        s0_pc <= s9_PCBranche;
      else
        s0_pc <= s1_pcPlusOne;
      end if;
    end if;
  end if;
end if;
end process;

-- Choose what address to go to FlashRAM ( PC or instrMemWAddr )
flashAddrMux: process( s0_pc )
begin
  if instrMemWE='1' then
    instrAddr(13 downto 0) <= instrMemWAddr(13 downto 0);
  else
    instrAddr <= s0_pc(15 downto 0);
  end if;
end process;

flash_mem_addr_buff <= "ZZZZZZZZZZZZZZZZ" when enable='0' else
  flash_mem_addr when enable='1' else
  "ZZZZZZZZZZZZZZZZ";
flash_ce_41 <= 'Z' when reset='1' else '0' when reset='0'
else 'Z';
flash_oe_43 <= 'Z' when reset='1' else '0' when reset='0'
else 'Z';
flash_a17_56 <= 'Z' when reset='1' else '0' when reset='0'
else 'Z';
--flash_a16_63 <= 'Z' when reset='1' else '0' when reset='0'
else 'Z';
flash_we_58 <= 'Z' when reset='1' else '1' when reset='0'
else 'Z';
flash_reset_59 <= 'Z' when reset='1' else '1' when reset='0'
else 'Z';

-- fetch instruction
instr_mem: mem_flash port map(instrAddr, clk10x, s1_instruction, flash_data_out, flash_mem_addr, reset );

-- fetch-decode separator
sep12_instr: bit16_reg port map( clk, enable, s1_instruction, s2_instruction );
sep12_pc:  bit16_reg port map( clk, true_bit, s1_pcPlusOne, s2_pcPlusOne );

-- decode
decoder: stage_decode port map( s2_instruction, s3_branch, s3_regWE, s3_PCResSel, s3_memWE, s3_flagWrtMask, s3_regWrtSel, s3_condCode, s3_isNegate, s3_ALUop, s3_resSel,
s3_muxA, s3_muxB, s3_immed, s3_shiftA, s3_ZSextendImmed, s3_selRegB, s3_selRegA );

-- decode-read separator
sep34_selRegA: bit4_reg port map( clk, enable, s3_selRegA, s4_selRegA );
sep34_selRegB: bit4_reg port map( clk, enable, s3_selRegB, s4_selRegB );
sep34_ZSexten: bit1_reg port map( clk, enable, s3_ZSextendImmed, s4_ZSextendImmed );
sep34_shiftA: bit2_reg port map( clk, enable, s3_shiftA,
sep34_imm: bit8_reg_port_map clk, enable, s3_imm,
s4_imm);  
sep34_muxA: bit2_reg_port_map clk, enable, s3_muxA, s4_muxA
);  
sep34_muxB: bit2_reg_port_map clk, enable, s3_muxB, s4_muxB
);  
sep34_resSel: bit2_reg_port_map clk, enable, s3_resSel,
s4_resSel);  
sep34_ALUop: bit2_reg_port_map clk, enable, s3_ALUop,
s4_ALUop);  
sep34_isNegat: bit1_reg_port_map clk, enable, s3_isNegate,
s4_isNegate);  
sep34_condCod: bit4_reg_port_map clk, enable, s3_condCode,
s4_condCode);  
sep34_flagWM: bit5_reg_port_map clk, enable, s3_flagWrtMask,
s4_flagWrtMask);  
sep34_branch: bit1_reg_port_map clk, enable, s3_branch,
s4_branch);  
sep34_regWE: bit1_reg_port_map clk, enable, s3_regWE,
s4_regWE);  
sep34_memWE: bit1_reg_port_map clk, enable, s3_memWE,
s4_memWE);  
sep34_PCResSel: bit1_reg_port_map clk, enable, s3_PCResSel,
s4_PCResSel);  
sep34_regWtSel: bit4_reg_port_map clk, enable, s3_regWrtSel,
s4_regWrtSel);  
sep34_pc: bit16_reg_port_map clk, enable, s2_pcPlusOne,
s4_pcPlusOne);

-- register file
regs: RegFile_port_map s4_selRegA, s4_selRegB, s8_regWrtSel,
s8_regWE, clk, reset,
s9_regWrtData, s5_regA,
s5_regB, reg00);

-- memory & IO
memAddrMux: mux2_port_map s5_regB, s8_memWrtAddr, s8_memWE,
s9_memAddr);  
io: io_controller_port_map reset, clk10x, s8_result,
s9_memAddr, s8_memWE, s5_memVal,
  vga_Hsync_p23, vga_Vsync_p26, vga_red1_p13,
vga_red0_p12, vga_grel1_p20, vga_gre0_p19, vga_blu1_p22,
vga_blu0_p21, nes_data_p80, nes_latch_p78,
nes_pulse_p94,
sound_bits_p87_to_p84);

-- read-op separator
sep56_ZSext: bit1_reg_port_map clk, enable, s4_ZSextendImm,
s6_ZSextendImm);  
sep56_shiftA: bit2_reg_port_map clk, enable, s4_shiftA,
s6_shiftA);  
sep56_imm: bit8_reg_port_map clk, enable, s4_imm,
s6_imm);  
sep56_muxA: bit2_reg_port_map clk, enable, s4_muxA, s6_muxA
);
sep56_muxB:  bit2_reg port map ( clk, enable, s4_muxB, s6_muxB );
sep56_resSel: bit2_reg port map ( clk, enable, s4_resSel, s6_resSel );
sep56_ALUop:  bit2_reg port map ( clk, enable, s4_ALUop, s6_ALUop );
sep56_isNegate: bit1_reg port map ( clk, enable, s4_isNegate, s6_isNegate );
sep56_condCod: bit4_reg port map ( clk, enable, s4_condCode, s6_condCode );
sep56_flagWM:  bit5_reg port map ( clk, enable, s4_flagWrtMask, s6_flagWrtMask );
sep56_branch:  bit1_reg port map ( clk, enable, s4_branch, s6_branch );
sep56_regWE:  bit1_reg port map ( clk, enable, s4_regWE, s6_regWE );
sep56_memWE:  bit1_reg port map ( clk, enable, s4_memWE, s6_memWE );
sep56_PCResSel: bit1_reg port map ( clk, enable, s4_PCResSel, s6_PCResSel );
sep56_regWSE: bit4_reg port map ( clk, enable, s4_regWrtSel, s6_regWrtSel );
sep56_pc:    bit16_reg port map ( clk, enable, s4_pcPlusOne, s6_pcPlusOne );
sep56_regA:  bit16_reg port map ( clk, enable, s5_regA, s6_regA );
sep56_regB:  bit16_reg port map ( clk, enable, s5_regB, s6_regB );
sep56_memVal: bit16_reg port map ( clk, enable, s5_memVal, s6_memVal );

-- op
op: stage_op port map ( s6_condCode, s6_prevFlags, s6_regA, s6_regB, s6_ZSextendImmed,
s6_immed, s6_pcPlusOne, s6_shiftA, s6_memVal, s6_resSel, s6_muxB, s6_muxA,
s6_aluOp, s6_isNegate, s7_flags, s7_result, s7_flagResultBit );

-- flag register
flagRegister:  flag_reg port map ( clk, s6_flagWrtMask, s7_flags, s6_prevFlags );

-- op-write separator
sep78_branch:  bit1_reg port map ( clk, enable, s6_branch, s8_branch );
sep78_regWE:  bit1_reg port map ( clk, enable, s6_regWE, s8_regWE );
sep78_memWE:  bit1_reg port map ( clk, enable, s6_memWE, s8_memWE );
sep78_PCResSel:  bit1_reg port map ( clk, enable, s6_PCResSel, s8_PCResSel );
sep78_regWSE:  bit4_reg port map ( clk, enable, s6_regWrtSel, s8_regWrtSel );
sep78_pc:    bit16_reg port map ( clk, enable, s6_pcPlusOne, s8_pcPlusOne );
sep78_result: bit16_reg port map( clk, enable, s7_result, s8_result);
sep78_flagBt: bit1_reg port map( clk, enable, s7_flagResultBit, s8_flagResultBit );
sep78_memWrtAddr: bit16_reg port map( clk, enable, s6_regB, s8_memWrtAddr );

-- write
regWrtDataMux: mux2 port map ( s8_pcPlusOne, s8_result, s8_PCRessSel, s9_regWrtData );
s9_pcBranch <= s8_result;
s9_PCSel <= s8_branch and s8_flagResultBit;

end pipeline_synth_arch;
**VHDL TestBench Programs**

**NOTE!** Our VHDL Testbench code is over 10,000 lines of VHDL code. If we were to print all of it we would extend the size of this report by 151 pages. It is highly unlikely that anyone would ever read all of that and if they did they would probably wish the whole time that they could just look at the code on the computer with a text editor and grep. Therefore we have only printed a small portion of our Testbench code to give the reader a feel for our style and we have enclosed a CD that contains all of VHDL Testbench code.

test_alu.vhd

```vhdl
-- Ben Holt    bholt@eng.utah.edu
-- Justin Olson justino@cs.utah.edu
-- Russell Christensen rchrist@cs.utah.edu
-- Usit Duongsaa duongsaa@cs.utah.edu
-- CS3710

-- Component : ComponentName

-- Person Responsible : Person(s) name (in case you have question, who knows it best)? Russ

-- What does it do? Tests the whole ALU

library ieee;
use ieee.std_logic_1164.all;

use std.textio.all;
use ieee.std_logic_textio.all;

entity test_alu is
end test_alu;

architecture implementation of test_alu is
component alu_16
port ( 
   op1     : in std_logic_vector(15 downto 0);  -- 1rst operand
   op2     : in std_logic_vector(15 downto 0);  -- 2nd operand
   isNegate : in std_logic;  -- Need to negate for
   subtraction
```
ALUop : in std_logic_vector(1 downto 0); -- Op code for operation
result : out std_logic_vector(15 downto 0);
-C, L, F, Z, N : out std_logic;
clfzn_flags : out std_logic_vector(4 downto 0));
end component;

-- purpose: Tests expected against output and reports if there is an error
procedure ReportCase (constant caseNum : in natural;
            expected, actual, first_input, second_input : in
std_logic_vector(15 downto 0);
            expected_flags, flags : in std_logic_vector(4 downto 0)) is
variable s : line;
begin -- ReportCase

if expected /= actual then
    write(s, string"("TestCase ")");
    write(s, caseNum, right, 3);
    write(s, string" Actual="));
    write(s, actual);
    write(s, string" Expect="));
    write(s, expected);
    write(s, string"***** FAIL *****"));
    writeln(output, s);
else
    if (expected_flags(4) /= '-') and (expected_flags(4) /= flags(4))
then
    write(s, string"("TestCase ")");
    write(s, caseNum, right, 3);
    write(s, string" L flag failed ");
    write(s, string"Actual = ");
    write(s, flags(4)));
    write(s, string"***** FAIL *****"));
    writeln(output, s);
    elsif (expected_flags(3) /= '-') and (expected_flags(3) /= flags(3))
then
    write(s, string"("TestCase ")");
    write(s, caseNum, right, 3);
    write(s, string" F flag failed ");
    write(s, string"******* FAIL *******"));
    writeln(output, s);
    elsif (expected_flags(2) /= '-') and (expected_flags(2) /= flags(2))
then
    write(s, string"("TestCase ")");
    write(s, caseNum, right, 3);
    write(s, string" F flag failed ");
    write(s, string"******* FAIL *******"));
    writeln(output, s);
    elsif (expected_flags(1) /= '-') and (expected_flags(1) /=
flags(1))
then
    write(s, string"("TestCase ")");
    write(s, caseNum, right, 3);
    write(s, string" Z flag failed ");
    write(s, string"******* FAIL *******"));
    writeln(output, s);
else if (expected_flags(0) /= '-') and (expected_flags(0) /= flags(0)) then
  write(s, string('"TestCase #"'));
  write(s, caseNum, right, 3);
  write(s, string(' N flag failed '));
  write(s, string('****** FAIL ******'));
  writeln(output, s);
else
  --write(s, string(' !!!SUCCESS!!'));
end if;
end if;

end ReportCase;

signal caseNum : natural;
signal expected, result : std_logic_vector(15 downto 0);
signal first_input, second_input : std_logic_vector(15 downto 0);
signal isDone : boolean := false;
signal isNegate : std_logic;
signal ALUop : std_logic_vector(1 downto 0);
signal clk : std_logic := '0';
signal reset : std_logic := '0';
signal flags : std_logic_vector(4 downto 0);
signal expected_flags : std_logic_vector(4 downto 0);
constant period : time := 100 ns;

begin
  DU T : alu_16 port map (first_input, second_input, isNegate, ALUop,
result, flags);
  -- Keep clock ticking while !isDone
  ClkProcess: Process (isDone, clk)
  begin
    if not isDone then
      clk <= not clk after period / 2;
    end if;
  end process;  --ClkProcess

  process  
  variable s : line;
  begin
    write(s, string(" Do ADD Tests"));
    writeln(output, s);
    caseNum <= 0;
    first_input <= "0000000000000000";
    second_input <= "0000000000000000";
    expected <= "0000000000000000";
    isNegate <= '0';
    ALUop <= "00";  --ADD_OPC
    expected_flags <= "0-0--";  --flags: C, L, F, Z, N
    wait for period;
    ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags,flags);

    caseNum <= caseNum + 1;
    first_input <= "0000000000000000";
    second_input <= "0000000000000000";
    expected <= "0000000000000000";

  end process;
end
isNegate <= '1';
expected_flags <= "0010";          --flags: C, L, F, Z, N
ALUop <= "00";                        --ADD_OPCODE
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "0000000000000001";
second_input <= "0000000000000001";
expected <= "0000000000000010";
isNegate <= '0';
expected_flags <= "0-0--";          --flags: C, L, F, Z, N
ALUop <= "00";                        --ADD_OPCODE
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;            --3
first_input <= "0011111111111111";
second_input <= "0000000000000001";
expected <= "0100000000000000";
isNegate <= '0';
expected_flags <= "0-0--";          --flags: C, L, F, Z, N
ALUop <= "00";                        --ADD_OPCODE
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "0000000000000101";
second_input <= "0000000000000010";
expected <= "0000000000000011";
isNegate <= '1';
expected_flags <= "1000";           --flags: C, L, F, Z, N
ALUop <= "00";                        --ADD_OPCODE
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "1111111111111111";
second_input <= "0000000000000001";
expected <= "0000000000000000";
isNegate <= '0';
expected_flags <= "1-0--";           --flags: C, L, F, Z, N
ALUop <= "00";                        --ADD_OPCODE
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;            --6
first_input <= "0111111111111111";
second_input <= "0000000000000001";
expected <= "1000000000000000";
isNegate <= '0';
expected_flags <= "0-1--";           --flags: C, L, F, Z, N
ALUop <= "00"; --ADD_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;    --7
first_input <= "1000000000000000"
second_input <= "0000000000000001"
expected <= "0111111111111111"
isNegate <= '1';
expected_flags <= "10101";    --flags: C, L, F, Z, N
ALUop <= "00";                --ADD_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "0000000000000001";
second_input <= "1000000000000000"
expected <= "1000000000000000"
isNegate <= '1';
expected_flags <= "01100";    --flags: C, L, F, Z, N
ALUop <= "00";                --ADD_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

--write(s, "     Do AND op code Tests");
--writeline(output, s);

caseNum <= caseNum + 1;
first_input <= "0000000000000000"
second_input <= "0000000000000000"
expected <= "0000000000000000"
expected_flags <= "-----";    --flags: C, L, F, Z, N
ALUop <= "01";                --AND_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "1111111111111111"
second_input <= "0000000000000000"
expected <= "0000000000000000"
ALUop <= "01";                --AND_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input <= "0000000000000000"
second_input <= "1111111111111111"
expected <= "0000000000000000"
ALUop <= "01";                --AND_OP
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input, expected_flags, flags);
caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "01";  --AND_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input,expected_flags,flags);

--write(s, "  Do OR op code Tests");
--writeline(output,s);

caseNum <= caseNum + 1;
first_input  <= "0000000000011100";
second_input <= "1000100100010100";
expected    <= "0000000000010100";
ALUop       <= "01";  --AND_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input,expected_flags,flags);

caseNum <= caseNum + 1;
first_input  <= "0000000000000000";
second_input <= "0000000000000000";
expected    <= "0000000000000000";
ALUop       <= "10";  -- OR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input,expected_flags,flags);

caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "0000000000000000";
expected    <= "1111111111111111";
ALUop       <= "10";  -- OR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input,expected_flags,flags);

caseNum <= caseNum + 1;
first_input  <= "0000000000000000";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "10";  -- OR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input,
second_input,expected_flags,flags);

caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "10";  -- OR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input  <= "00000000000011100";
second_input <= "1000100100010100";
expected    <= "1000100100011100";
ALUop       <= "10";               -- OR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

--write(s, " Do XOR op code Tests");
--writeline(output,s);

caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "11";               --XOR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input  <= "0000000000000000";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "11";               --XOR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "11";               --XOR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

caseNum <= caseNum + 1;
first_input  <= "1111111111111111";
second_input <= "1111111111111111";
expected    <= "1111111111111111";
ALUop       <= "11";               --XOR_OPC
wait for period;
ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);

--write(s, " Do XOR op code Tests");
--writeline(output,s);

caseNum <= caseNum + 1;
first_input  <= "00000000000011100";
second_input <= "1000100100010100";
expected    <= "1000100100011100";
ALUop       <= "11";               --XOR_OPC
wait for period;
  ReportCase(caseNum, expected, result, first_input, second_input, expected_flags, flags);
  
isDone <= true;
  wait;
end process;
end implementation;
**CASM Test Code**

**NOTE!** Our CASM Test Code is over 2,400 lines of CASM code. If we were to print all of it we would extend the size of this report by 52 pages. It is highly unlikely that anyone would ever read all of that and if they did they would probably wish the whole time that they could just look at the code on the computer with a text editor and grep. Therefore we have only printed a small portion of our CASM Test Code to give the reader a feel for our style and we have enclosed a CD that contains all of our CASM Test Code.

test_jal.cpp
/* Ben Holt bholt@eng.utah.edu
-- Justin Olson justino@cs.utah.edu
-- Russell Christensen rchriste@cs.utah.edu
-- Usit Duongsa duongsa@cs.utah.edu
-- CS3710

-- Component : ComponentName
-- Person Responsible : Russ
-- What does it do?
*/

#include ".\tools\cr16lib.h"

// testcase #33

void cr16main()
{
    activateHazardDetectionUnit();
    compileAndSimulate(__FILE__);
    expectResult("0000000000001101");
    runFor(700);
    movi(0, r10);
    loadLabelAddress("label", r11); // 6 instructions
    movi(1, r0);
    movi(2, r1);
    nop();
    nop();
    jal(r10, r11); // r10 = 12


```cpp
test_scond_ge.cpp
/* Ben Holt                bholt@eng.utah.edu
   -- Justin Olson          justino@cs.utah.edu
   -- Russell Christensen  rchriste@cs.utah.edu
   -- Usit Duongsaa        duongsaa@cs.utah.edu
   -- CS3710

   -- Component : ComponentName
   -- Person Responsible : Russ
   -- What does it do?
   */

#include "../tools/cr16lib.h"

// testcase #11

void cr16main()
{
    activateHazardDetectionUnit();
    compileAndSimulate(__FILE__);
    expectResult("00000000000000110");
    runFor(700);
    movi(10, r0);
    movi(10, r1);
    movi(1, r2);
    cmp(r0, r1);
    sCond(GE, r0);
    lsh(r2, r0);
    mov(r0, r3);
    cmp(r1, r2);
    sCond(GE, r0);
    add(r3, r0);
    lshi(1, r0);
    mov(r0, r4);  // r4 = r0
    cmp(r2, r1);
    sCond(GE, r0);
    add(r4, r0);
    // expect r0 to be 6
```
fib_rec.cpp

// fib.cpp : Defines the entry point for the console application.
//

#include "../tools/cr16lib.h"
#include <iostream>
#include <string>

using namespace std;
static cr16_int sp(r15);
static cr16_int fp(r7);
static cr16_int ra(r4);

inline void fib()
{
    addLabel("fib");
    cr16_int res(r0);
    cr16_int n(r1);
    cr16_int call(r2);
    cr16_int retFrom(r3);
    // temp registers
    cr16_int dbg(r8);
    cr16_int temp(r9);

    sp += 2;        // allocate 2 words on stack
    fp = sp - 1;
    store(ra, fp);

    loadLabelAddress("return", retFrom);
    cr16_if (n == 0 | n == 1);
    {
        movi(1, res);
        jCond(UC, retFrom);flushPipeline();
    }
    cr16_else();
    {
        loadLabelAddress("fib", call);
        --n;

        store(n, sp);
        jal(ra, call); flushPipeline();
        load(n, sp);

        temp = res;
        --n;

        sp+=2;
fp = sp - 1;
store(n, fp);
fp = sp - 2;
store(temp, fp);
jal(ra, call); flushPipeline();
fp = sp - 1;
load(n, fp);
fp = sp - 2;
load(temp, fp);
sp-=2;

res += temp;
}
cr16_endif();

addLabel("return");
fp = sp - 1;
load(ra, fp);
sp -= 2;
//bCondLabel(UC, "EOF");
jCond(UC, ra); flushPipeline();
}

void cr16main()
{
 activateHazardDetectionUnit();
 compileAndSimulate(__FILE__);
 expectResult("000000011111000");
 runFor(40000);

 // init stack pointer r15
 // ret r14
 cr16_int arg1(r1);
cr16_int res(r0);
sp = 0;
arg1 = 11;
res = 0;

loadLabelAddress("fib", r2);
jal(ra, r2); flushPipeline();
loadLabelAddress("EOF", r2);
jCond( UC, r2 );flushPipeline();

fib();

addLabel("EOF");
flushPipeline();
}
**Show Off Applications Source Code**

**NOTE! We have not taken effort to prevent line wrapping.**
When developing these applications we have had a screen wider than this piece of paper. We considered reformatting our source code but while that would make it look nicer on paper it would make it look worse on the computer screen. So we include the source code of these programs on a CD attached with this report. If you are going to study this programs in depth you are strongly urged to go with the attached electronic version of this source code.

```
tetris.cpp
//Tetris for the CASM Processor
//Written by: Russ Christensen <rchriste@cs.utah.edu>

#include "../..tools/crl6lib.h"

#define FUNCTION_RETURN
   jCond(UC, ret_addr); \ flushPipeline()

//when speed tuning you will not want to use this macro because it will cause nops()
//between the two instructions that could be filled with useful instructions
#define FUNCTION_CALL(label)
   loadLabelAddress(label, r13); /*r13 is a scratch register*/ \ jal(ret_addr, r13);
   flushPipeline()

//Some simple macros to make saving registers to memory and loading them
//back easier. Please remember that these registers are being saved on //the stack. Which is a stack!!! Meaning you can only remove items in the
//reverse order that you place them on.
#define SAVE_VARIABLE(var)
   store(var,SP); \ ++SP

#define RESTORE_VARIABLE(var)
```

77
load(var, SP)

//returns number = number*10
//uses tmp as scratch space
#define MULT_10(var,tmp) 
    tmp = var + var;     /* tmp = 2*n */     
    var = tmp + tmp;     /* n = 4*n */     
    var += var;          /* n = 8*n */     
    var = var + tmp      /* n = 10*n */
#define MULT_8(var) 
    var += var;          /* var = 2*var */     
    var += var;          /* var = 4*var */     
    var += var;          /* var = 8*var */
#define MULT_2(var) 
    var += var /* var = 2*var */
#define DIV_2(var) 
    lshi(-1, var) /*var = var / 2 */
#define CR16_FOR(first,second,third,body) 
    { 
        cr16_while(second);       
        {                         
            body;                  
        }                       
        third;                  
    } cr16_endwhile()

//var = var*amount
//The Console Revolution 16 does not support multiplication in hardware
//so here is a software algorithm to get it done.
//If you know the amount at compile time then use one of the other
//macros that is specialized.
//This is a simple algorithm, I'm sure you could look up something
//much fancier in a book and use that instead. If I have some time
//to do some heavy optimizing then I will change this.
#define MULT(var, tmp, amount) 
    tmp = 0; 
    cr16_while(amount); 
    { 
        tmp = tmp + var; 
        --amount; 
    } cr16_endwhile(); 
    var = tmp

namespace {
    bool debug = false; //should debug info be compiled in
}

***************
How things are setup with the current color pallet
    const int BLACK = 0;
    const int RED = 1;
const int BLUE = 2;
const int GREEN = 3;
**************************************************************************/

namespace tetris_constants {
    const int A_BUTTON = 0x80;
    const int B_BUTTON = 0x40;
    const int SELECT_BUTTON = 0x20;
    const int START_BUTTON = 0x10;
    const int LEFT_BUTTON = 0x08;
    const int RIGHT_BUTTON = 0x04;
    const int UP_BUTTON = 0x02;
    const int DOWN_BUTTON = 0x01;
    const int SLIDE_MASK = 0x03;
    const int FALSE = 0;
    const int TRUE = 1;
    const int WIDTH = 64;
    const int HEIGHT = 30;
    const int TETRIS_WIDTH = 10;
    const int TETRIS_HEIGHT = 20;
    const int LEFT = 20; // left boarder line
    const int RIGHT = LEFT+TETRIS_WIDTH*2+1; // right boarder line
    const int STARTING_COUNT_MAX = 20;
    const int MIN_COUNT_MAX = 5; // lowest I ever want count_max to be
    const int TOP = 5;
    const int BOTTOM = TOP+TETRIS_HEIGHT; // bottom line
    // Please note that it is absolutely necessary for the correctness of this
    // program that the BACKGROUND_COLOR be 0 and the
    FALLEN_PIECE_COLOR be 3
    // I use bitwise operations that depend on this!!! This is a speed
    // optimization
    // PIECE_COLOR and BOARDER_COLOR can be exchanged.
    const int PIECE_COLOR = 1;
    const int BOARDER_COLOR = 2;
    const int BACKGROUND_COLOR = 0;
    const int FALLEN_PIECE_COLOR = 3;
    const int NES_ADDR = 0xE000; // Memory mapped address of NES
}

Controller
const int NUMBER_OF_TETRIS_PIECES = 7;
const int NUMBER_OF_ROTATIONS = 4;
const int BLOCKS_PER_PIECE = 4;
// 2 words (x position and y position) for each of the 4 blocks
that make up each tetris piece.
const int SPACE_FOR_EACH_ROTATION = BLOCKS_PER_PIECE * 2;
const int PIECES_LIST_LENGTH = 35;

const int SOUND_ENABLE = 0x08;
const int SOUND_BIT2 = 0x04;
const int SOUND_BIT1 = 0x02;
const int SOUND_BIT0 = 0x01;

const int ADDR_VGA_PIXELS = 0xe800;
const int ADDR_VGA_VSYNC = 0xf800;
const int ADDR_VGA_FLIP = 0xf000;
const int ADDR_NES = 0xe000;
const int ADDR_SOUND = 0xd800;

cr16_int SP(r15); //stack pointer
    cr16_int ret_addr(r14); //return address for function calls

using namespace tetris_constants;

namespace memory_map {
    /***map in the memory address of some global variables***/
    //game_board is an array that contains where fallen tetris pieces lay.
    //it would be possible to use one bit per unit rather than one word
    //however memory is not my limitation speed is and this way will give me
    //[slightly] better performance
    const int game_board_start = 0;
    const int game_board_end = TETRIS_WIDTH*TETRIS_HEIGHT+1; //one past the end
    //memory to store data on what each peice looks like. This memory will be initialized
    //with constant values in the main function of the game
    const int game_pieces_start = game_board_end;
    const int game_pieces_end = game_pieces_start + NUMBER_OF_TETRIS_PIECES * NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION + 1; //one past the end
    const int current_piece = game_pieces_end; //only needs one word
    const int static_y = current_piece + 1;
    const int current_piece_list_position = static_y + 1;
    const int lines_distrroyed = current_piece_list_position + 1;
    //number of lines the player has distrroyed in the game
    const int count_max = lines_distrroyed + 1;
    const int lines_were_distrroyed = count_max + 1;
    const int pieces_list = lines_were_distrroyed + 1;
    const int BEGIN_STACK = pieces_list + PIECES_LIST_LENGTH+1;
}

namespace game_piece {
    //map out the memory addresses of each tetris peice. The actual memory is mapped in with
    //game_pieces_start to game_peices_end.
    const int square = memory_map::game_pieces_start;
    const int line = square + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
    const int left_L = line + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
    const int right_L = left_L + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
    //don't know what to call the last peice so I call it foo
    const int left_foo = right_L + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
    const int right_foo = left_foo + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
    const int mountain = right_foo + NUMBER_OF_ROTATIONS * SPACE_FOR_EACH_ROTATION;
}
namespace inline_function {
    void flipPage();
}

namespace sound_inline_function {
    void inline_sound_effect0() {
        cr16_int addr(r9);
        cr16_int value(r8);
        SAVE_VARIABLE(r8);
        SAVE_VARIABLE(r9);

        addr = tetris_constants::ADDR_SOUND;
        store(value, addr);

        RESTORE_VARIABLE(r9);
        RESTORE_VARIABLE(r8);
    }

    void inline_sound_effect1() {
        cr16_int addr(r6);
        cr16_int value(r7);
        SAVE_VARIABLE(r6);
        SAVE_VARIABLE(r7);

        addr = tetris_constants::ADDR_SOUND;
        value = SOUND_BIT0 | SOUND_BIT1 | SOUND_BIT2 | SOUND_ENABLE;
        store(value, addr);

        CR16_FOR(value = 0, value < 10, ++value,
        {
            inline_function::flipPage();
        });

        inline_function::flipPage();

        RESTORE_VARIABLE(r7);
        RESTORE_VARIABLE(r6);
    }

    void inline_sound_off() {
        cr16_int addr(r9);
        cr16_int value(r8);
        SAVE_VARIABLE(r8);
        SAVE_VARIABLE(r9);

        addr = ADDR_SOUND;
        load(value, addr);
        value = 0;
        store(value, addr);

        RESTORE_VARIABLE(r9);
        RESTORE_VARIABLE(r8);
    }
}
//Put all the inline functions in a namespace to make the code more readable
//to those who don't know CASM inside out.
namespace inline_function {
    //The following functions are inline functions that are automatically
    //expanded by CASM at compile time. Most of tetris is inline functions
    //because I get better performance doing this way. I don't have to pay
    //the [small, but present] speed price of a function call. And because
    //our Console Revolution 16 processor has no cache the only
    //drawback to making everything inline is code size. Except so long
    //as the program fits inside of 64k words code size is not an issue either
} //---------------------------------

//An inline function that changes the graphics buffer
void flipPage() {
    cr16_int addr(r8), data(r9);

    // wait until Vsync starts
    data = 0;
    addr = 0xf800;
    cr16_while( data==0 );
    {
        load( data, addr );
        flushPipeline();
    } cr16_endwhile();

    // Flip page
    addr = 0xf000;
    data = 0;
    store( data, addr );
    flushPipeline();
    flushPipeline();
    flushPipeline();
    flushPipeline();
    data = 1;
    store( data, addr );
    flushPipeline();
}

//---------------------------------

//inline function that plots a point
void drawPixel( cr16_int x, cr16_int y, cr16_int color ) {
    cr16_int addr(r8);
    cr16_int vga_addr(r9);
    vga_addr = ADDR_VGA_PIXELS;
    addr = y+y;    // 2*y
    addr += addr;  // 4*y
    addr += addr;  // 8*y
vga_addr += x;
addr += addr;    // 16*y
addr += addr;    // 32*y
addr += addr;    // 64*y
addr += vga_addr;
store( color, addr );
}

//----------------------------------------------------------
-
// print content of reg9 (between 0..999)
void printReg9()
{
    SAVE_VARIABLE(r0);
    SAVE_VARIABLE(r1);
    SAVE_VARIABLE(r2);
    SAVE_VARIABLE(r3);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r8);

    loadLabelAddress( "skipDrawCode", r1 );
    jCond( UC, r1 );
    flushPipeline();

    // Code that draw the digit 0,1,2...9
    ifstream f("numbers.pix");
    // read image data of all digits
    vector<int> x[10], y[10];
    float z;
    for( int j=29; j>=0; j-- )
        for( int i=0; i<64; i++ )
            {
                f >> z;
                if( z<0.5 )
                    {
                        int which = i/6;
                        assert( which>=0 && which<10 );
                        // separate them into 10 sets (one for each digit)
                        x[which].push_back(i-which*6);
                        y[which].push_back(j);
                    }
            }
    for( int which=0; which<10; which++ )
        {
            addLabel( "drawDigitLabel" + int2bin(which,16) );
            r0 = ( ADDR_VGA_PIXELS + y[which][0]*64 + x[which][0] ) +
                 r4;
            r1 = ( ADDR_VGA_PIXELS + y[which][1]*64 + x[which][1] ) +
                 r4;
            r2 = ( ADDR_VGA_PIXELS + y[which][2]*64 + x[which][2] ) +
                 r4;
            r3 = 1;
            unsigned int index = 3, size = x[which].size();
        }
while( index < size )
{
    store( r3, r0 );
    store( r3, r1 );
    store( r3, r2 );
    r0 += (y[which][index]*64+x[which][index] - y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size) break;
    r1 += (y[which][index]*64+x[which][index] - y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size) break;
    r2 += (y[which][index]*64+x[which][index] - y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size) break;
}
store( r3, r0 );
store( r3, r1 );
store( r3, r2 );
jCond(UC,r6);  // return
flushPipeline();

flightline غالباً ما يكون مرتبطًا مع 

// r0 = digit to draw, r4=address offset
addLabel("DrawDigit");
cr16_if( r0==0 ); loadLabelAddress( "drawDigitLabel" + int2bin(0,16), r1 ); cr16_endif();
cr16_if( r0==1 ); loadLabelAddress( "drawDigitLabel" + int2bin(1,16), r1 ); cr16_endif();
cr16_if( r0==2 ); loadLabelAddress( "drawDigitLabel" + int2bin(2,16), r1 ); cr16_endif();
cr16_if( r0==3 ); loadLabelAddress( "drawDigitLabel" + int2bin(3,16), r1 ); cr16_endif();
cr16_if( r0==4 ); loadLabelAddress( "drawDigitLabel" + int2bin(4,16), r1 ); cr16_endif();
cr16_if( r0==5 ); loadLabelAddress( "drawDigitLabel" + int2bin(5,16), r1 ); cr16_endif();
cr16_if( r0==6 ); loadLabelAddress( "drawDigitLabel" + int2bin(6,16), r1 ); cr16_endif();
cr16_if( r0==7 ); loadLabelAddress( "drawDigitLabel" + int2bin(7,16), r1 ); cr16_endif();
cr16_if( r0==8 ); loadLabelAddress( "drawDigitLabel" + int2bin(8,16), r1 ); cr16_endif();
cr16_if( r0==9 ); loadLabelAddress( "drawDigitLabel" + int2bin(9,16), r1 ); cr16_endif();
jal( r6, r1 );  // call draw
flushPipeline();
jCond(UC,r14);  // return
flushPipeline();

// determines what three digits to draw
addLabel("skipDrawCode");

r0 = 0;  // the 'hundred' digit
cr16_while(r9 >= 100 );
r9 -= 100;
+++r0;
flushPipeline();
cr16_endwhile();
r4 = 57 + (64*0);
loadLabelAddress( "DrawDigit", r1 );
jal(r14,r1);
flushPipeline();

r0 = 0;            // the 'ten'
digit
cr16_while(r9 >= 10 );
  r9 -= 10;
  ++r0;
  flushPipeline();
cr16_endwhile();
r4 = 57 + (64*9);
loadLabelAddress( "DrawDigit", r1 );
jal(r14,r1);
flushPipeline();

r0 = r9;            // the 'one' digit
r4 = 57 + (64*18);
loadLabelAddress( "DrawDigit", r1 );
jal(r14,r1);
flushPipeline();

RESTORE_VARIABLE(r8);
RESTORE_VARIABLE(r7);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r3);
RESTORE_VARIABLE(r2);
RESTORE_VARIABLE(r1);
RESTORE_VARIABLE(r0);

// -------------------------------
// inline function that will print the given register on the upper left hand
// side of the screen on the second row down.  value is assumed to be in
// register r0
void debug_printValue(crl6_int value)
{
  crl6_int x(r7);
  crl6_int y(r6);
  crl6_int color(r5);
  crl6_int addr(r4);
  SAVE_VARIABLE(r4);
  SAVE_VARIABLE(r5);
  SAVE_VARIABLE(r6);
  SAVE_VARIABLE(r7);
  addr = memory_map::static_y;
load(y, addr);
++y;
cr16_if(y > 30);
{
    y = 1;
} cr16 endif();
store(y, addr);

for(int i=0; i<2; ++i) {
    SAVE_VARIABLE(value);
    // put a different color on both sides of the readout to
    make it easier to read
    x=1;
    color = 3;
    drawPixel(x, y, color);
    CR16_FOR(x=2, x < 18, ++x,
    {
        color = value & 1,
        lshi(-1, value),
        drawPixel(x, y, color)
    });
    color = 3;
    drawPixel(x, y, color);
    RESTORE_VARIABLE(value);
    flipPage();
}

RESTORE_VARIABLE(r9);
RESTORE_VARIABLE(r8);
RESTORE_VARIABLE(r7);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r4);

//-----------------------------------------------
-
// sets background to BACKGROUND_COLOR
void setBackground()
{
    cr16_int x(r7);
    cr16_int y(r6);
    cr16_int color(r5);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);

    x = 0;
    color = BACKGROUND_COLOR;
    cr16_while(x < WIDTH);
    {
        y=0;
        cr16_while(y < HEIGHT);
        {
            drawPixel(x, y, color);
            ++y;
        } cr16 endwhile();
        ++x;
    } cr16 endwhile();
}
} cr16_endwhile();

RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);

//sets background to BACKGROUND_COLOR
void clearScore()
{
    cr16_int x(r7);
    cr16_int y(r6);
    cr16_int color(r5);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);

    x = 56;
    color = BACKGROUND_COLOR;
    cr16_while(x < WIDTH);
    {
        y = 0;
        cr16_while(y < HEIGHT);
        {
            drawPixel(x, y, color);
            ++y;
        } cr16_endwhile();
        ++x;
    } cr16_endwhile();

    RESTORE_VARIABLE(r5);
    RESTORE_VARIABLE(r6);
    RESTORE_VARIABLE(r7);
}

//inlined function that draws the tetris boarder
void inlined drawBorder()
{
    cr16_int y(r7);
    cr16_int x(r6);
    cr16_int color(r5);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);

    //draw side lines
    color = BOARDER_COLOR;
    y = TOP;
    cr16_while(y < BOTTOM);
    {
        x = LEFT;
        drawPixel(x, y, color);
        --x;
        drawPixel(x, y, color);
    }
x = RIGHT;
drawPixel(x,y,color);
++x;
drawPixel(x,y,color);
++y;
} cr16_endif();
//draw bottom line
x = LEFT-1;
y = BOTTOM;
cr16_while( x <= RIGHT + 1);
{
    drawPixel(x,y,color);
    ++y;
    drawPixel(x,y,color);
    --y;
    ++x;
} cr16_endif();

RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
}

//inline function that returns true if the position given is valid
void checkCurrentPiece(cr16_int result, cr16_int x, cr16_int y,
cr16_int rotation)
{
    cr16_int tmp(r8);
    cr16_int addr(r7);
    cr16_int x_value(r6);
    cr16_int y_value(r5);
    cr16_int current_piece_ptr(r4);
    SAVE_VARIABLE(r8);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(rotation);
    SAVE_VARIABLE(x);
    SAVE_VARIABLE(y);

    //checkSquare(result,x,y,rotation);

    result = TRUE;
    addr = memory_map::current_piece;
    load(current_piece_ptr, addr);
    rotation = rotation & 3; //just get the lower 2 bits of the
rotation
    assert(SPACE_FOR_EACH_ROTATION == 8 && "Must modify line below so
the mult is correct");
    MULT_8(rotation);
    current_piece_ptr += rotation;
    //First check to see if the piece will remain in the tetris box
    SAVE_VARIABLE(current_piece_ptr);
    for(int i=0; i < BLOCKS_PER_PIECE; ++i)
    {
        load(x_value, current_piece_ptr);

++current_piece_ptr;
load(y_value, current_piece_ptr);
++current_piece_ptr;
MULT_2(x_value);
x_value += x;
y_value += y;

//@todo This is weird behavior I don't understand why I have to have a minus 4 and a minus 2
  cr16_if((x_value < LEFT) | (x_value > RIGHT - 2) | (y_value > BOTTOM - 1));
  {
    result = FALSE;
  } cr16_endif();
}
//Now check to see if the space is filled on the game board
RESTORE_VARIABLE(current_piece_ptr);
for(int i=0; i < BLOCKS_PER PIECE; ++i)
{
  load(x_value, current_piece_ptr);
  ++current_piece_ptr;
  MULT_2(x_value);
  load(y_value, current_piece_ptr);
  ++current_piece_ptr;
  x_value += x;
  y_value += y;
  x_value = x_value-LEFT;
  DIV_2(x_value);
  y_value = y_value-TOP;
  MULT_10(y_value,tmp);
  addr = x_value + y_value;

  /////it is possible to get invalid input so check for that
  //if (debug) {
  //  SAVE_VARIABLE(r0);
  //  r0 = addr;
  //  debug_printValue(r0);
  //  RESTORE_VARIABLE(r0);
  //}
  cr16_if((addr >= 0) & (addr < memory_map::game_board_end));
  {
    load(tmp,addr);
    cr16_if(tmp == FALLEN_PIECE_COLOR);
    {
      result = FALSE;
    } cr16_endif();
  } cr16_endif();
}

RESTORE_VARIABLE(y);
RESTORE_VARIABLE(rotation);
RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);
RESTORE_VARIABLE(r8);
void placeCurrentPiece(cr16_int x, cr16_int y, cr16_int rotation) {
    cr16_int tmp(r8);
    cr16_int addr(r7);
    cr16_int x_value(r6);
    cr16_int y_value(r5);
    cr16_int current_piece_ptr(r4);
    SAVE_VARIABLE(r8);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(rotation);
    SAVE_VARIABLE(x);
    SAVE_VARIABLE(y);

    addr = memory_map::current_piece;
    load(current_piece_ptr, addr);
    rotation = rotation & 3; // just get the lower 2 bits of the rotation
    assert(SPACE_FOR_EACH_ROTATION == 8 && "Must modify line below so the mult is correct");
    MULT_8(rotation);
    current_piece_ptr += rotation;
    for(int i=0; i < BLOCKS_PER PIECE; ++i) {
        load(x_value, current_piece_ptr);
        ++current_piece_ptr;
        MULT_2(x_value);
        load(y_value, current_piece_ptr);
        ++current_piece_ptr;
        x_value += x;
        y_value += y;
        x_value = x_value - LEFT;
        DIV_2(x_value);
        y_value = y_value - TOP;
        MULT_10(y_value, tmp);
        addr = x_value + y_value;
        tmp = FALLEN_PIECE_COLOR;
        cr16_if((addr >= 0) & (addr < memory_map::game_board_end)) {
            store(tmp, addr);
        } cr16_else();
        // signal an error by halting the program
        if(debug) {
            loadLabelAddress("waitForStart", r0);
            jCond(UC, r0);
            flushPipeline();
        }
    } cr16 endif();
}

RESTORE_VARIABLE(y);
void drawCurrentPiece(cr16_int x, cr16_int y, cr16_int rotation)
{
    cr16_int addr(r7);    /* NOTE!! Union */
    cr16_int color(r7);    /* NOTE!! Union */
    cr16_int x_value(r6);
    cr16_int y_value(r5);
    cr16_int current_piece_ptr(r4);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(rotation);

    addr = memory_map::current_piece;
    load(current_piece_ptr, addr);
    rotation = rotation & 3; //just get the lower 2 bits of the
    rotation
    assert(SPACE_FOR_EACH_ROTATION == 8 && "Must modify line below so
    the mult is correct");
    MULT_8(rotation);
    current_piece_ptr += rotation;
    for(int i=0; i < BLOCKS_PER_PIECE; ++i)
    {
        load(x_value, current_piece_ptr);
        ++current_piece_ptr;
        load(y_value, current_piece_ptr);
        MULT_2(x_value);
        x_value += x;
        y_value += y;
        ++current_piece_ptr;
        color = PIECE_COLOR;
        inline_function::drawPixel(x_value, y_value, color);
        ++x_value;
        inline_function::drawPixel(x_value, y_value, color);
    }
}

//restore all variables

void drawFallenPieces()
{
    //because this function doesn't take any arguments it is okay to
/use the lower registers, even though that is against my convention
    cr16_int pixel_x(r7);
cr16_int pixel_y(r6);
cr16_int pixel_color(r5);
cr16_int addr(r4);
cr16_int data(r3);
cr16_int x(r2);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(r3);
    SAVE_VARIABLE(r2);
    pixel_color = FALLEN_PIECE_COLOR;
    //the for loop is preprocessor loop unrolling
    for(int y=0; y < TETRIS_HEIGHT; ++y) {
        x = 0;
cr16_while(x < TETRIS_WIDTH);
        {
            //I can get away with a multiplication
            //because it is collapsed to a constant at compile time
            addr = (y*TETRIS_WIDTH)+x;
            load(data,addr);
            pixel_x = LEFT+1+x+x;
            pixel_y = TOP+y;
drawPixel(pixel_x,pixel_y,data);
++pixel_x;
drawPixel(pixel_x,pixel_y,data);
++x;
        } cr16_endwhile();
    }
    RESTORE_VARIABLE(r2);
    RESTORE_VARIABLE(r3);
    RESTORE_VARIABLE(r4);
    RESTORE_VARIABLE(r5);
    RESTORE_VARIABLE(r6);
    RESTORE_VARIABLE(r7);
}
//Get a new game piece
void changeCurrentPiece()
{
    cr16_int piece_list_index(r7);
cr16_int piece(r6);
cr16_int addr(r5);
cr16_int tmp(r4);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
//This code reads the piece out of a list
addr = memory_map::current_piece_list_position;
load(piece_list_index, addr);
++piece_list_index;
cr16_if(piece_list_index >= PIECES_LIST_LENGTH);
{
    piece_list_index = 0;
} cr16_endif();
store(piece_list_index, addr);

tmp = memory_map::pieces_list;
addr = piece_list_index + tmp;
load(piece, addr);
addr = memory_map::current_piece;
store(piece, addr);

///This code cycles through the pieces
//addr = memory_map::current_piece;
//load(piece, addr);
//piece += tetrisGlobals::SPACE_FOR_EACH_ROTATION *
tetris_globals::NUMBER_OF_ROTATIONS;
//cr16_if(piece > game_piece::right_foo);
//{
//    piece = game_piece::square;
//} cr16_endif();
//store(piece, addr);

RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);

// Declare the global data
void initialize_global_data()
{
    cr16_int addr(r0);
    cr16_int part0_x(r1);
    cr16_int part0_y(r2);
    cr16_int part1_x(r3);
    cr16_int part1_y(r4);
    cr16_int part2_x(r5);
    cr16_int part2_y(r6);
    cr16_int part3_x(r7);
    cr16_int part3_y(r8);
    cr16_int tmp(r9);
    SAVE_VARIABLE(r9);
    SAVE_VARIABLE(r8);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(r3);
    SAVE_VARIABLE(r2);
    SAVE_VARIABLE(r1);
    SAVE_VARIABLE(r0);
{  //initialize memory_map
    cr16_int i(r8);  //i is being treated as a pointer below
    cr16_int zero(r9);

    zero = 0;
    CR16_FOR(i = memory_map::game_board_start, i <
            memory_map::game_board_end, ++i,
            ...
( //becuase of how this is implemented must use '(' in
store(zero,i)
}); //end CR16_FOR

///@todo also make this function do the inializing of the
//game_data memory area

//load the memory address of the square peice
addr = game_piece::square;
//set the x and y value of all four tetris peice parts in all
rotations
for(int i=0; i < NUMBER_OF_ROTATIONS; ++i)
{
    part0_x = 0;
    part0_y = 0;
    part1_x = 1;
    part1_y = 0;
    part2_x = 0;
    part2_y = 1;
    part3_x = 1;
    part3_y = 1;
    store(part0_x, addr);
    ++addr;
    store(part0_y, addr);
    ++addr;
    store(part1_x, addr);
    ++addr;
    store(part1_y, addr);
    ++addr;
    store(part2_x, addr);
    ++addr;
    store(part2_y, addr);
    ++addr;
    store(part3_x, addr);
    ++addr;
    store(part3_y, addr);
    ++addr;
}

addr = game_piece::line;
//rotations 0 & 2 and 1 & 3 are the same
for(int i=0; i < NUMBER_OF_ROTATIONS; i += 2)
{
    //rotations 0 & 2
    part0_x = 1;
    part0_y = 0;
    part1_x = 1;
    part1_y = 1;
    part2_x = 1;
    part2_y = 2;
    part3_x = 1;
    part3_y = 3;
    store(part0_x, addr);
    ++addr;
    store(part0_y, addr);
    ++addr;
    store(part1_x, addr);
}
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

// rotations 1 & 3
part0_x = 0;
part0_y = 0;
part1_x = 1;
part1_y = 0;
part2_x = 2;
part2_y = 0;
part3_x = 3;
part3_y = 0;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

addr = game_piece::left_foo;
// rotations 0 & 2 and 1 & 3 are the same
for(int i=0; i < NUMBER_OF_ROTATIONS; i += 2)
{
    // rotations 0 & 2
part0_x = 0;
part0_y = 0;
part1_x = 1;
part1_y = 0;
part2_x = 1;
part2_y = 1;
part3_x = 2;
part3_y = 1;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

//rotations 1 & 3
part0_x = 1;
part0_y = 0;
part1_x = 0;
part1_y = 1;
part2_x = 1;
part2_y = 1;
part3_x = 0;
part3_y = 2;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

)  
addr = game_piece::right_foo;
//rotations 0 & 2 and 1 & 3 are the same
for(int i=0; i < NUMBER_OF_ROTATIONS; i += 2) 
{
   //rotations 0 & 2
   part0_x = 1;
   part0_y = 0;
   part1_x = 2;
   part1_y = 0;
   part2_x = 0;
   part2_y = 1;
   part3_x = 1;
   part3_y = 1;
   store(part0_x, addr);
   ++addr;
   store(part0_y, addr);
   ++addr;
   store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

// rotations 1 & 3
part0_x = 0;
part0_y = 0;
part1_x = 0;
part1_y = 1;
part2_x = 1;
part2_y = 1;
part3_x = 1;
part3_y = 2;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;
}

addr = game_piece::left_L;
// All rotations are different
{
    // rotation 0
    part0_x = 0;
    part0_y = 0;
    part1_x = 1;
    part1_y = 0;
    part2_x = 1;
    part2_y = 1;
    part3_x = 1;
    part3_y = 2;
    store(part0_x, addr);
    ++addr;
    store(part0_y, addr);
    ++addr;
    store(part1_x, addr);
    ++addr;
store(part1_y, addr);
  ++addr;
store(part2_x, addr);
  ++addr;
store(part2_y, addr);
  ++addr;
store(part3_x, addr);
  ++addr;
store(part3_y, addr);
  ++addr;

  //rotation 1
  part0_x = 0;
  part0_y = 0;
  part1_x = 1;
  part1_y = 0;
  part2_x = 2;
  part2_y = 0;
  part3_x = 0;
  part3_y = 1;
store(part0_x, addr);
  ++addr;
store(part0_y, addr);
  ++addr;
store(part1_x, addr);
  ++addr;
store(part1_y, addr);
  ++addr;
store(part2_x, addr);
  ++addr;
store(part2_y, addr);
  ++addr;
store(part3_x, addr);
  ++addr;
store(part3_y, addr);
  ++addr;

  //rotation 2
  part0_x = 0;
  part0_y = 0;
  part1_x = 0;
  part1_y = 1;
  part2_x = 0;
  part2_y = 2;
  part3_x = 1;
  part3_y = 2;
store(part0_x, addr);
  ++addr;
store(part0_y, addr);
  ++addr;
store(part1_x, addr);
  ++addr;
store(part1_y, addr);
  ++addr;
store(part2_x, addr);
  ++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

// rotation 3
part0_x = 0;
part0_y = 1;
part1_x = 1;
part1_y = 1;
part2_x = 2;
part2_y = 1;
part3_x = 2;
part3_y = 0;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

}
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

//rotation 1
part0_x = 0;
part0_y = 0;
part1_x = 0;
part1_y = 1;
part2_x = 1;
part2_y = 1;
part3_x = 2;
part3_y = 1;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

//rotation 2
part0_x = 1;
part0_y = 0;
part1_x = 1;
part1_y = 1;
part2_x = 0;
part2_y = 2;
part3_x = 1;
part3_y = 2;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;
//rotation 3
part0_x = 0;
part0_y = 0;
part1_x = 1;
part1_y = 0;
part2_x = 2;
part2_y = 0;
part3_x = 2;
part3_y = 1;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;
}

addr = game_piece::mountain;
//All rotations are different
{
    //rotation 0
    part0_x = 1;
    part0_y = 0;
    part1_x = 0;
    part1_y = 1;
    part2_x = 1;
    part2_y = 1;
    part3_x = 2;
    part3_y = 1;
    store(part0_x, addr);
    ++addr;
    store(part0_y, addr);
    ++addr;
    store(part1_x, addr);
    ++addr;
    store(part1_y, addr);
    ++addr;
    store(part2_x, addr);
    ++addr;
    store(part2_y, addr);
    ++addr;
    store(part3_x, addr);
    ++addr;
    store(part3_y, addr);
    ++addr;
}
//rotation 1
part0_x = 2;
part0_y = 0;
part1_x = 1;
part1_y = 1;
part2_x = 2;
part2_y = 1;
part3_x = 2;
part3_y = 2;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

//rotation 2
part0_x = 0;
part0_y = 0;
part1_x = 1;
part1_y = 0;
part2_x = 2;
part2_y = 0;
part3_x = 1;
part3_y = 1;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;

//rotation 3
part0_x = 1;
part0_y = 0;
part1_x = 1;
part1_y = 1;
part2_x = 2;
part2_y = 1;
part3_x = 1;
part3_y = 2;
store(part0_x, addr);
++addr;
store(part0_y, addr);
++addr;
store(part1_x, addr);
++addr;
store(part1_y, addr);
++addr;
store(part2_x, addr);
++addr;
store(part2_y, addr);
++addr;
store(part3_x, addr);
++addr;
store(part3_y, addr);
++addr;
}

//initialize the list of pieces, rather than random numbers this is the list that we draw from
{
cr16_int value(r1);
SAVE_VARIABLE(r1);

assert(PIECES_LIST_LENGTH == 35 && "Initalize the values for a list size of 30");
addr = memory_map::pieces_list;
value = game_piece::square;
store(value, addr);
++addr;
value = game_piece::left_L;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;
value = game_piece::line;
store(value, addr);
++addr;
value = game_piece::left_L;
store(value, addr);
++addr;
value = game_piece::right_foo;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;
value = game_piece::square;
store(value, addr);
++addr;
value = game_piece::left_L;
store(value, addr);
++addr;
value = game_piece::right_L;
store(value, addr);
++addr;
value = game_piece::left_foo;
store(value, addr);
++addr;
value = game_piece::left_L;
store(value, addr);
++addr;
value = game_piece::square;
store(value, addr);
++addr;
value = game_piece::line;
store(value, addr);
++addr;
value = game_piece::line;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;
value = game_piece::left_foo;
store(value, addr);
++addr;
value = game_piece::right_foo;
store(value, addr);
++addr;
value = game_piece::right_L;
store(value, addr);
++addr;
value = game_piece::square;
store(value, addr);
++addr;
value = game_piece::square;
store(value, addr);
++addr;
value = game_piece::left_foo;
store(value, addr);
++addr;
value = game_piece::right_L;
store(value, addr);
++addr;
value = game_piece::right_foo;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;
value = game_piece::left_L;
store(value, addr);
++addr;
value = game_piece::right_foo;
store(value, addr);
++addr;
value = game_piece::line;
store(value, addr);
++addr;
value = game_piece::right_L;
store(value, addr);
++addr;
value = game_piece::right_foo;
store(value, addr);
++addr;
value = game_piece::left_foo;
store(value, addr);
++addr;
value = game_piece::right_L;
store(value, addr);
++addr;
value = game_piece::line;
store(value, addr);
++addr;
value = game_piece::left_foo;
store(value, addr);
++addr;
value = game_piece::mountain;
store(value, addr);
++addr;

RESTORE_VARIABLE(r1);
}

inline_function::changeCurrentPiece();
tmp = 0;
addr = memory_map::static_y;
store(tmp, addr);
tmp = 0;
addr = memory_map::lines_destroyed;
store(tmp, addr);
tmp = tetris_constants::STARTING_COUNT_MAX;
addr = memory_map::count_max;
store(tmp, addr);

RESTORE_VARIABLE(r0);
RESTORE_VARIABLE(r1);
RESTORE_VARIABLE(r2);
RESTORE_VARIABLE(r3);
RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);
RESTORE_VARIABLE(r8);
RESTORE_VARIABLE(r9);

//-------------------------------
-

//inline function that prints what buttons are being pressed down in the
void debug_print_controller()
{
    //because this function takes no arguments it is okay to use lower registers even though that is against programming convention
    cr16_int tmp(r7);
    cr16_int bit(r6);
    cr16_int x(r5);
    cr16_int y(r4);
    cr16_int nes_buttons(r3);
    SAVE_VARIABLE(r7);
    SAVE_VARIABLE(r6);
    SAVE_VARIABLE(r5);
    SAVE_VARIABLE(r4);
    SAVE_VARIABLE(r3);

    tmp = 0x0E00;
    load(nes_buttons,tmp);
    x = 1;
    y = 0;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
    bit = nes_buttons;
    andi(1, bit);
    drawPixel(x,y, bit);
    lshl(-1,nes_buttons);
    ++x;
bit = nes_buttons;
andi(1, bit);
drawPixel(x, y, bit);
lsli(-1, nes_buttons);
++x;

RESTORE_VARIABLE(r3);
RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);
}

// writes a checkered pattern to the screen
void debug_setCheckeredBackground()
{
  cr16_int x(r7);
  cr16_int y(r6);
  cr16_int color(r5);
  SAVE_VARIABLE(r7);
  SAVE_VARIABLE(r6);
  SAVE_VARIABLE(r5);

  x = 0;
  cr16_while(x < WIDTH);
  {
    y = 0;
    cr16_while(y < HEIGHT);
    {
      // don't need to modulus color because the hardware
      // only looks at the bottom two bits
      color = x + y;
      drawPixel(x, y, color);
      ++y;
    } cr16_endwhile();
    ++x;
  } cr16_endwhile();

  RESTORE_VARIABLE(r5);
  RESTORE_VARIABLE(r6);
  RESTORE_VARIABLE(r7);
}

// -------------------------------------------------------------------------------
void cr16main()
{
    //if (debug)    cerr << "Stack Begins at " << BEGIN_STACK << endl;
    activateHazardDetectionUnit();

    //main function
    addLabel("main");
    {
        //The convention that I will use is R0-R3 will NEVER be used
        //by any inline functions that take arguments because these
        //registers might be passed in as data. Otherwise it is only
        //okay for [real] functions and inline functions that don’t
        //take arguments to use these registers just so long as the
        //called function saves and restores the register.

        //This convention is necessary because while CASM supports
        //inline functions it does not support automatic variable to
        //register mapping and as a result when you use inline
        //functions you must be careful.
        cr16_int x(r0);
        cr16_int y(r1);
        cr16_int rotation(r2);
        cr16_int result(r3);

        //These are registers that are local variables that will never
        //be passed into inline functions. Called functions must save
        //away these registers and restore them so I can assume that
        //they always have the correct value
        cr16_int nes_buttons(r4);
        cr16_int rot_down(r5); //@todo could use the high bits of
        rotation
        cr16_int count(r6);
        cr16_int color(r7);

        //registers R8-9 are assumed to be destroyed when a function call
        //is made
        //registers R10-R13 is reserved for the CASM compiler
        //register R14 is to store the return address for a function call
        //because I have so much data memory (64k words) I
        //almost
        //always inline function calls so I don't use this
        much
        //register R15 is for the stack pointer
SP = memory_map::BEGIN_STACK;

// Most of the time I just do functions inline, but this is an example of how an actual function call would be made.
FUNCTION_CALL("initVGA");

// This is an inline function at compile time it is expanded to be the full function
inline_function::flipPage();

// Initialize game data
rotation = 0;
count = 0;
color = BACKGROUND_COLOR;
rl = 0; // Beginning memory address of game_data array
CR16_FOR(rl=0, rl < TETRIS_WIDTH * TETRIS_HEIGHT, ++rl,
{
    store(color, rl)
});

// for(int i=0; i < TETRIS_WIDTH * TETRIS_HEIGHT; ++i) {
// // Remember that this for loop happens at compile time, it can be thought of
// // // as a preprocessor directive so what I am doing here
// // // is loop unrolling at compile time.
// // store(color, rl);
// // ++rl;
// //}
CR16_FOR(rl=0, rl < 2, ++rl,
{
    inline_function::setBackground(),
    inline_function::drawBorder(),
    inline_function::flipPage()
});

addLabel("waitForStart");
cr16_while(1);
{
    cr16_int tmp(r8);

    sound_inline_function::sound_off();
    inline_function::changeCurrentPiece(); // so we get a random starting piece

    tmp = NES_ADDR;
    load(nes_buttons, tmp);
    cr16_if(nes_buttons & START_BUTTON);
    {
        loadLabelAddress("startGame", tmp);
        sound_inline_function::sound_off();
        jCond(UC, tmp);
        flushPipeline();
    } cr16_endif();
} cr16_endwhile();

addLabel("startGame");
{
    // for loop happens at compile time, this is loop unrolling
    for(int i=0; i < 2; ++i)
Once for each memory buffer:
inline_function::setBackground();
inline_function::drawBorderColor();
inline_function::flipPage();

// initialze data that I am treating as global constants
inline_function::initialize_global_data();

y = TOP;
x = LEFT+9;
count = 0;

Main Game Loop

addLabel("mainGameLoop");
{
cr16_int tmp_addr(r8);
cr16_int tmp(r9);

if (debug) { // debug should be thought of as a preprocessor directive
    inline_function::debug_print_controller();
}

tmp = NES_ADDR;
load(nes_buttons,tmp);

cr16_if(nes_buttons & RIGHT_BUTTON);  
{
x = x+2;
inline_function::checkCurrentPiece(result,x,y,rotation);
    cr16_if((result == FALSE) | (count & SLIDE_MASK));
    {
x = x-2;
    } cr16_endif();
} cr16_endif();
cr16_if(nes_buttons & LEFT_BUTTON);  
{
x = x-2;
inline_function::checkCurrentPiece(result,x,y,rotation);
    cr16_if(result == FALSE | (count & SLIDE_MASK));
    {
x = x+2;
    } cr16_endif();
} cr16_endif();
cr16_if(nes_buttons & UP_BUTTON);  
{
    // @todo Right now UP_BUTTON does nothing. I think later on up should
//drop the piece
} else {
  ++y;
}

inline_function::checkCurrentPiece(result, x, y, rotation);
  cr16_if(result == FALSE);
  {
    --y;
  } else {
    count = 0;
  }
} else {
  count = 0;
}

//@todo can probably take these two statements out of
the if block
//@todo the next statement is probably not needed
rot_down = rot_down & (A_BUTTON | B_BUTTON);
rot_down = rot_down & (nes_buttons & (A_BUTTON | B_BUTTON));
}

else {
  {
    cr16_if(nes_buttons & B_BUTTON);
    {
      ++rotation;
      rot_down = nes_buttons;
    }
    inline_function::checkCurrentPiece(result, x, y, rotation);
    cr16_if(result == FALSE);
    {
      --rotation;
      rot_down = 0;
    }
  }
} else {
  {
    cr16_if(nes_buttons & A_BUTTON);
    {
      --rotation;
      rot_down = nes_buttons;
    }
  }
} else {
  inline_function::checkCurrentPiece(result, x, y, rotation);
  cr16_if(result == FALSE);
  {
    ++rotation;
    rot_down = 0;
  }
} else {
  inline_function::checkCurrentPiece(result, x, y, rotation);
  cr16_if(result == FALSE);
  {
    --rotation;
    rot_down = nes_buttons;
  }
}

//stop the game
loadLabelAddress("waitForStart", r0);
jCond(UC, r0);
flushPipeline();
}
tmp_addr = memory_map::count_max;
load(tmp, tmp_addr);
cr16_if(count > tmp + MIN_COUNT_MAX);
{
  ++y;
}
inline_function::checkCurrentPiece(result, x, y, rotation);
cr16_if(result == FALSE);
{
  --y;
}
inline_function::placeCurrentPiece(x, y, rotation);
    // destroyLines must be a real function call
    because it might call itself
FUNCTION_CALL("destroyLines");
y = TOP;
x = LEFT + 9;
rotation = 0;
inline_function::checkCurrentPiece(result, x, y, rotation);
cr16_if(result == FALSE);
{
  SAVE_VARIABLE(r0);
  loadLabelAddress("waitForStart", r0);
  jCond(UC, r0);
  RESTORE_VARIABLE(r0);  // this will still happen because we have 4 instructions
    // after a jump before the jump takes effect
    flushPipeline();
} cr16 endif();
inline_function::changeCurrentPiece();
cr16 endif();
} cr16 endif();
cr16_if(count < 3);
{
  sound_inline_function::sound_effect0();
} cr16_else();
{
  sound_inline_function::sound_off();
} cr16 endif();
++count;
inline_function::drawFallenPieces();
inline_function::drawCurrentPiece(x, y, rotation);
inline_function::flipPage();
loadLabelAddress("mainGameLoop", tmp);
jCond(UC, tmp);
flushPipeline();
}  // end main game loop

// end the program
loadLabelAddress("theEnd", r0);
jCond(UC, r0);
flushPipeline();
}

// -----------------------------------------------------------------------------
-
// Destroys complete lines at the bottom of the board.
// like a good tetris game should :)
// This function takes no arguments
addLabel("destroyLines");
{
  cr16_int i(r7);
  cr16_int addr(r6);
  cr16_int tmp(r5);
  cr16_int lines_were_destroyed(r4);
  cr16_int function_call_ColumnToLookAt(r1);
  cr16_int result(r0);
  SAVE_VARIABLE(ret_addr);
  SAVE_VARIABLE(r7);
  SAVE_VARIABLE(r6);
  SAVE_VARIABLE(r5);
  SAVE_VARIABLE(r4);
  SAVE_VARIABLE(r3);
  SAVE_VARIABLE(r2);
  SAVE_VARIABLE(r1);
  SAVE_VARIABLE(r0);

  tmp = FALSE;
  addr = memory_map::lines_were_destroyed;
  store(tmp, addr);
  i = TETRIS_HEIGHT-1;
  cr16_while(i >= 0);
  {
    function_call_ColumnToLookAt = i;
    FUNCTION_CALL("lookAndDestroySingleLine");
    cr16_while(result);
    {
      addr = memory_map::lines_were_destroyed;
      load(tmp, addr);
      tmp = TRUE;
      store(tmp, addr);
      addr = memory_map::lines_destroyed;
      load(tmp, addr);
      ++tmp;
      store(tmp, addr);
      addr = memory_map::count_max;
      load(tmp, addr);
      --tmp;
      cr16_if(tmp < 0);
      {
        tmp = 0;
      } cr16 endif();
      store(tmp, addr);
      function_call_ColumnToLookAt = i;
      FUNCTION_CALL("lookAndDestroySingleLine");
    } cr16 endwhile();
  } --i;
}
} cr16_endwhile();

addr = memory_map::lines_were_destroyed;
load(tmp, addr);
cr16_if(tmp);
{
    CR16_FOR(i=0, i < 2, ++i,
    {
        inline_function::clearScore(),
        addr = memory_map::lines_destroyed,
        load(r9, addr),
        inline_function::printReg9(),
        inline_function::flipPage()
    });
    sound_inline_function::sound_effect1();
} cr16 endif();

RESTORE_VARIABLE(r0);
RESTORE_VARIABLE(r1);
RESTORE_VARIABLE(r2);
RESTORE_VARIABLE(r3);
RESTORE_VARIABLE(r4);
RESTORE_VARIABLE(r5);
RESTORE_VARIABLE(r6);
RESTORE_VARIABLE(r7);
RESTORE_VARIABLE(ret_addr);
FUNCTION_RETURN;

//----------------------------------------------------------------------------------------------------------------------

//THIS FUNCTION TAKES PARAMETERS AND HAS A RETURN VALUE!!!!!!
//This function returns whether it destroyed a line in r0
addLabel("lookAndDestroySingleLine");
{
    //Arguments passed in:
    cr16_int ColumnToLookAt(r1);

    //Return Value
    cr16_int result(r0);
    cr16_int base_addr(r2);
    cr16_int addr(r3);
    cr16_int data(r4);
    cr16_int tmp(r5);

    result = FALSE;
    base_addr = TETRIS_WIDTH;
    MULT(base_addr, tmp, ColumnToLookAt); //base_addr = TETRIS_WIDTH
    * ColumnToLookAt
    data = FALLEN_PIECE_COLOR;
    addr = base_addr;
    //this for loop is CASM compile time loop unrolling
    for(int i=0; i < TETRIS_WIDTH; ++i)
    {
        load(tmp, addr);
        ++addr;
        data = data & tmp;

crl6_if(data);
{
    // redefine variable names for more readable code, because these registers are also defined below this could best be though of as a C union
    crl6_int source_addr(r3);
    crl6_int dest_addr(r4);
    crl6_int data(r5);

    result = TRUE; // TRUE we did destroy a line
    // we have a complete row that needs to be destroyed. Copy all the rows above down a row
    source_addr = base_addr - 1;
    dest_addr = source_addr + TETRIS_WIDTH;
    crl6_while(source_addr > TETRIS_WIDTH);
    {
        load(data, source_addr);
        store(data, dest_addr);
        --source_addr;
        --dest_addr;
    } crl6 endwhile();
    data = BACKGROUND_COLOR;
    crl6_while(source_addr >= 0);
    {
        store(data, source_addr);
        --source_addr;
    } crl6 endwhile();
    // see if there is another row to destroy. This could be done as tail // recursion and probably should be. I just want to show that CASM and our processor is // capable of true recursion. There is a limit of 3k of data memory if that is exceeded // the behavior of Console Revolution 16 is undefined, however we will never get that high // in this case because there is a limited number of rows and columns
} crl6 endif();
FUNCTION_RETURN;
}

//--------------------------------------------------------------
// You must call this function before using VGA
addLabel("initVGA");
{
    crl6_int addr(r8), data(r9);
    // if I was going to make any other function calls inside of this function then I would need to push ret_addr onto the stack and restore
    // it right before the FUNCTION_RETURN call

    // setup first page
    addr = 0xf000;
    data = 1;
    store(data, addr);


```cpp
FUNCTION_RETURN;

// have a label to the end of the program because CASM automatically
// infinite loop at the bottom of the program.
addLabel("theEnd");
}

invader.cpp
/
-- Ben Holt        bholt@eng.utah.edu
-- Justin Olson    justino@cs.utah.edu
-- Russell Christensen  rchriste@cs.utah.edu
-- Usit Duongsa    duongsa@cs.utah.edu
-- CS3710

-- Component: Invader

-- Person Responsible: Usit

-- The Space Invaders Game
*/

#include "..\..\tools\cr16lib.h"

//-----------------------------------------------------------------------------
const int black = 0;
const int red = 1;
const int blu = 2;
const int gre = 3;

const int NES_A = 0x80;
const int NES_B = 0x40;
const int NES_SEL = 0x20;
const int NES_START = 0x10;
const int NES_LEFT = 0x08;
const int NES_RIGHT = 0x04;
const int NES_UP = 0x02;
const int NES_DOWN = 0x01;

const int ADDR_VGA_PIXELS = 0xe800;
const int ADDR_VGA_VSYNC = 0xf800;
const int ADDR_VGA_FLIP = 0xf000;
const int ADDR_NES = 0xe000;
const int ADDR_SOUND = 0xd800;

const int ALIEN1 = 1;
const int ALIEN2 = 2;
const int ALIEN3 = 3;
const int ALIEN4 = 4;
const int HEALTH_EXIST = 10;    // alien with
health below this amount is not there at all
```
const int HEALTH_ALIVE = 20; // alien with health below this amount is "exploding"

when health reaches 0, it disappears // if
health>HEALTH_MIN, alien is still alive // if
health%2==1, it should flashes

const int NEXT_BULLET = 2; // the size of a bullet record is 2, so to get the next bullet, we +2
const int NEXT_ALIEN = 4;
const int NEXT_PARTICLE = 4;
const int NEXT_MISSILE = 3;
const int NEXT_STAR = 3;

const int NUM_Bullets = 5;
const int NUM_ALIENS = 10;
const int NUM_PARTICLES = 32;
const int NUM_MISSILES = 8;
const int NUM_STARS = 14;

const int SOUND_NONE = 0;
const int SOUND_PLAYER_FIRE = 1;
const int SOUND_ALIEN_DIE = 2;

enum mem_map // Memory Address Scheme for the SRAM
{
    mem_last_input = 0, // input code from last cycle
cycle
    mem_this_input = 1, // this
been pressed this cycle
mem_once_input = 2, // buttons that have just
mem_rand = 3, // last random number
mem_tick = 4, // number of ticks (cycles)
since reset (wraps around 16-bit)
mem_sound_type = 5,
mem_sound_remaining = 6,

// player's ship
mem_ship_x = 10,
mem_ship_y = 11,
mem_ship_alive = 12,
mem_score = 15,

// a list of 5 bullets, each represented as (x,y)
mem_bullets_head = 100,
mem_bullets_tail = 109,

// a list of 10 aliens, each represented as (health,x,y,type)
mem_aliens_head = 110,
mem_aliens_tail = 149,

// a list of 32 particles, each represented as x,y,dx,dy
mem_particles_head = 150,
mem_particles_tail = 277,
mem_particle_slot = 278,  // next time we add a
  particle, add it here

// a list of 8 alien missiles, each represented as x,y,age
mem_missiles_head = 280,
mem_missiles_tail = 303,

// a list of 14 stars, each represented as x,y,speedRate
mem_stars_head = 310,
mem_stars_tail = 351,
}

// void memLoad( cr16_int var, int addr )
// { 
//   r11 = addr;
//   load( var, r11 );
// }

// void memSave( cr16_int var, int addr )
// {
//   r11 = addr;
//   store( var, r11 );
// }

// void memInit( int value, int addr )
// {
//   r11 = addr;
//   r12 = value;
//   store( r12, r11 );
// }

// // returns a random number between 0..255
// void getRand( cr16_int result )
// {
//   memLoad( result, mem_rand );
//   result += 317;
//   cr16_if( result > 5413 )
//   result -= 5413;
//   cr16 endif();
// 
//   //r10 = 3719;
//   //xor( r10, result );
//   memSave( result, mem_rand );
//   
//   result = result & 0x0ff;
// }

// void flipPage()
cr16_int addr(r8), data(r9);

// wait until Vsync starts
data = 0;
addr = ADDR_VGA_VSYNC;
cr16_while( data==0 );
{
    load( data, addr );
    flushPipeline();
}

// Flip page
addr = ADDR_VGA_FLIP;
data = 0;
store( data, addr );
flushPipeline();
flushPipeline();
flushPipeline();
data = 1;
store( data, addr );
flushPipeline();

// void initVGA()
{
    cr16_int addr(r8), data(r9);
    
    // setup first page
    addr = ADDR_VGA_FLIP;
data = 1;
store( data, addr );
flushPipeline();
flushPipeline();
flushPipeline();
flushPipeline();
}

// void updateTick()
{
    memLoad( r0, mem_tick );
++r0;
    memSave( r0, mem_tick );
}

// void playSound( int soundType )
{
    cr16_int oldType(r8), newType(r9);
memLoad( oldType, mem_sound_type );
newType = soundType;
cr16_if( (newType+1) >= (oldType+1) );
{ 
cr16_int duration(r8);
void initGameData()
{
    // system
    memInit( 0, mem_this_input );
    memInit( 0, mem_tick );

    // player's ship
    memInit( 32, mem_ship_x );
    memInit( 27, mem_ship_y );
    memInit( 1, mem_ship_alive );
    memInit( 10, mem_score );

    // clear bullets list
    memInit( 0, mem_bullets_head+1 );
    memInit( 0, mem_bullets_head+3 );
    memInit( 0, mem_bullets_head+5 );
    memInit( 0, mem_bullets_head+7 );
    memInit( 0, mem_bullets_head+9 );

    // clear aliens
    memInit( 0, mem_aliens_head +NEXT_ALIEN*0 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*1 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*2 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*3 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*4 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*5 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*6 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*7 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*8 );
    memInit( 0, mem_aliens_head +NEXT_ALIEN*9 );

    // clear particles
    memInit( 0, mem_particle_slot );
    cr16_int index(r0), addr(r1), x(r2);
    index = 0;
    addr = mem_particles_head;
    x = 0;
    cr16_while( index < 40 )
    {
        store( x, addr );
        ++index;
        addr += NEXT_PARTICLE;
    }
    cr16 endwhile();

    // clear alien missiles
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*0 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*1 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*2 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*3 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*4 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*5 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*6 );
memInit( 30, mem_missiles_head+1+NEXT_MISSILE*7 );

// init stars
for( int i=0; i<NUM_STARS; i++ )
{
    int x = (rand()%51)+6;
    int y = (i*7)%30;
    int sr = i%2;   // speed rating
    if(sr==0) sr=3;
    memInit( x, mem_stars_head+0+i*NEXT_STAR );
    memInit( y, mem_stars_head+1+i*NEXT_STAR );
    memInit( sr, mem_stars_head+2+i*NEXT_STAR );
}

// clear sound buffer
memInit( 0, mem_sound_type );
memInit( 0, mem_sound_remaining );
memInit( 0x0000, ADDR_SOUND );
}

// upward/upright/right/downright/down/downleft/left/upleft particles
void createExplosion( cr16_int x, cr16_int y )
{
    // must only uses reg9.
    cr16_int ptrA(r9), ptrB(r10), plus(r11), minus(r12), zero(r13);
    memLoad( ptrA, mem_particle_slot );
    assert(NEXT_PARTICLE==4);
    ptrB = mem_particles_head;
    ptrA += ptrA;   // 2x
    zero   = 0;
    plus   = 1;
    ptrA += ptrA;   // 4x
    minus  = zero;
    ptrA += ptrB;
    -minus;
    ptrB = ptrA + 1;
    // upw daright/downright/down/downleft/left/upleft
    store( x, ptrA ); store( y, ptrB ); ptrA += 2; ptrB += 2;
    store( zero  , ptrA ); store( minus, ptrB ); ptrA += 2; ptrB += 2;
    store( x, ptrA ); store( y, ptrB ); ptrA += 2; ptrB += 2;
    store( plus  , ptrA ); store( minus, ptrB ); ptrA += 2; ptrB += 2;
    store( x, ptrA ); store( y, ptrB ); ptrA += 2; ptrB += 2;
    store( plus  , ptrA ); store( zero  , ptrB ); ptrA += 2; ptrB += 2;
    store( x, ptrA ); store( y, ptrB ); ptrA += 2; ptrB += 2;
    store( zero  , ptrA ); store( plus  , ptrB ); ptrA += 2; ptrB += 2;
store(x, ptrA); store(y, ptrB); ptrA += 2; ptrB += 2;
store(minus, ptrA); store(plus, ptrB); ptrA += 2; ptrB += 2;
store(x, ptrA); store(y, ptrB); ptrA += 2; ptrB += 2;
store(minus, ptrA); store(zero, ptrB); ptrA += 2; ptrB += 2;
store(minus, ptrA); store(minus, ptrB); ptrA += 2; ptrB += 2;

memLoad(ptrA, mem_particle_slot);
ptrA += 8;
ptrA = ptrA & 0x001f;
memSave(ptrA, mem_particle_slot);
}

/------------------------------------------------------------------------

void drawPixel(cr16_int x, cr16_int y, cr16_int color)
{
    cr16_int addr(r8);
cr16_int vga_addr(r9);
vga_addr = ADDR_VGA_PIXELS;
addr = y+y; // 2*y
addr += addr; // 4*y
addr += addr; // 8*y
vga_addr += x;
addr += addr; // 16*y
addr += addr; // 32*y
addr += addr; // 64*y
addr += vga_addr;
store(color, addr);
}

/------------------------------------------------------------------------

void drawPixelStatic(int x, int y, int color)
{
    int address = (y*64+x) | ADDR_VGA_PIXELS;
r10 = address;
r11 = color;
    store(r11, r10);
}

/------------------------------------------------------------------------

void clearBackground()
{
    cr16_int color(r7);
cr16_int addr(r8), stop(r9);
addr = ADDR_VGA_PIXELS;
stop = (64*30);
    color = black;
stop += addr;
cr16_while(addr<stop);
{
    store(color, addr);
    ++addr;
}
    cr16 endwhile();
}

/------------------------------------------------------------------------

125
void drawBorder()
{
    for ( int y=0; y<30; y++ )
    {
        drawPixelStatic( 6,y, gre );
        drawPixelStatic( 57,y, gre );
    }
}

//---------------------------------------------------------------

void drawShip( cr16_int x, cr16_int y, cr16_int colGre, cr16_int colBlu )
{
    cr16_int addr1(r10);
    cr16_int addr2(r11);
    cr16_int offset(r11);
    addr1 = y;
    offset = ADDR_VGA_PIXELS;
    addr1 += addr1;  // 2y
    addr1 += addr1;  // 4y
    offset += x;
    addr1 += addr1;  // 8y
    addr1 += addr1;  // 16y
    offset -= 128;     // 2 rows up
    addr1 += addr1;  // 32y
    addr1 += addr1;  // 64y

    addr1 += offset;
    addr2 = addr1 + 63;
    store( colBlu, addr1 );
    store( colGre, addr2 );
    addr1 += 64;
    addr2 += 2;
    store( colBlu, addr1 );
    store( colGre, addr2 );
    addr1 += 62;
    addr2 += 62;
    store( colGre, addr1 );
    store( colBlu, addr2 );
    addr1 += 2;
    addr2 += 2;
    store( colBlu, addr1 );
    store( colBlu, addr2 );
    addr1 += 2;
    addr2 += 62;
    store( colGre, addr1 );
    store( colGre, addr2 );
    addr1 += 63;
    store( colGre, addr1 );
}

//---------------------------------------------------------------

void drawAlien1( cr16_int x, cr16_int y, cr16_int colRed, cr16_int colBlu )
{
    cr16_int addr1(r10);
    cr16_int addr2(r11);
crl6_int offset(r11);
addr1 = y;
offset = ADDR_VGA_PIXELS;
addr1 += addr1; // 2y
addr1 += addr1; // 4y
offset += x;
addr1 += addr1; // 8y
addr1 += addr1; // 16y
offset -= 66; // 1 row up, 2 col left
addr1 += addr1; // 32y
addr1 += addr1; // 64y

cr16_int offset(r11);
addr1 = y;
offset = ADDR_VGA_PIXELS;
addr1 += addr1; // 2y
addr1 += addr1; // 4y
offset += x;
addr1 += addr1; // 8y
addr1 += addr1; // 16y
offset -= 65; // 1 row up, 1 col left
addr1 += addr1; // 32y
addr1 += addr1; // 64y

cr16_int addr1(r10);
cr16_int addr2(r11);
cr16_int offset(r11);
addr1 = y;
offset = ADDR_VGA_PIXELS;
addr1 += addr1; // 2y
addr1 += addr1; // 4y
offset += x;
addr1 += addr1; // 8y
addr1 += addr1; // 16y
offset -= 65; // 1 row up, 1 col left
addr1 += addr1; // 32y
addr1 += addr1; // 64y

addr1 += offset;
addr2 = addr1 + 1;
store( colBlu, addr1 );
store( colRed, addr1 );
addr1 += 2;
addr2 += 62;
store( colBlu, addr1 );
store( colBlu, addr2 );
addr1 += 62;
addr2 += 2;
store( colRed, addr1 );
store( colRed, addr2 );
addr1 += 2;
addr2 += 2;
store( colRed, addr1 );
store( colBlu, addr2 );
addr1 += 61;
addr2 += 64;
store( colBlu, addr1 );
store( colBlu, addr2 );
}
//----------------------------------------------------------
-
void drawAlienMissle( cr16_int x, cr16_int y, cr16_int age )
{
    cr16_int color(r9);
    cr16_int addr1(r10);
    cr16_int addr2(r11);
    cr16_int offset(r11);
    addr1 = y;
    offset = ADDR_VGA_PIXELS;
    addr1 += addr1;  // 2y
    addr1 += addr1;  // 4y
    offset += x;
    addr1 += addr1;  // 8y
    addr1 += addr1;  // 16y
    addr1 += addr1;  // 32y
    addr1 += addr1;  // 64y
    color = age & 0x0003;
    addr1 += offset;
    addr2 = addr1;
    addr1 -= 64;    // 1 row up
    store( color, addr1 );
    store( color, addr1 );
    addr1 += 63;
    addr2 += 1;
    store( color, addr1 );
    store( color, addr2 );
    addr1 += 65;
    store( color, addr1 );
}
//----------------------------------------------------------
-
void drawAliens()
{
    cr16_int listPtr(r0), health(r1), x(r2), y(r3), type(r4);
    cr16_int colA(r5), colB(r6);

    listPtr = mem.aliens_head;
    colA = red;

    cr16_while( listPtr < mem.aliens_tail );
    {
        load( health, listPtr );
        cr16_if( health > HEALTH_EXIST );
        {
            r10 = listPtr + 1;
            r11 = listPtr + 2;
        }
r12 = listPtr + 3;
load( x, r10 );
load( y, r11 );
load( type, r12 );
cr16_if( health < HEALTH_ALIVE | health&0x0==1 );
    colB = red;
cr16_else();
    colB = blu;
cr16_endif();

cr16_if( type==ALIEN1 );
drawAlien1(x,y,colA,colB);
cr16_else();
cr16_if( type==ALIEN2 );
drawAlien2(x,y,colA,colB);  cr16_endif();
cr16_endif();
}
cr16_endwhile();
}
void drawBullets()
{
cr16_int listPtrX(r0), listPtrY(r1), x(r2), y(r3), colRed(r4);
listPtrY = (mem_bullets_head+1);
listPtrY += NEXT_BULLET;
    load( y, listPtrY );
    listPtrX = listPtrY - 1;
cr16_if( y > 0 );
    load( x, listPtrX );
    drawPixel( x,y, colRed );
}  cr16_endif();
}
void drawMissiles()
{
cr16_int listPtrX(r0), listPtrY(r1), listPtrAge(r2);
  cr16_int x(r3), y(r4), age(r5);
listPtrX = mem_missiles_head;
listPtrY = 1;
listPtrAge = 2;
listPtrY += listPtrX;
listPtrAge += listPtrX;
cr16_while( listPtrX <= mem_missiles_tail );
{
    load( y, listPtrY );
    load( x, listPtrX );
    load( age, listPtrAge );
    cr16_if( y < 30 ); // if active missile
        drawAlienMissile( x,y,age );
    cr16_endif();
    listPtrX   += NEXT_MISSILE;
    listPtrY   += NEXT_MISSILE;
    listPtrAge += NEXT_MISSILE;
}  cr16_endwhile();
}  //----------------------------------------------------------------------

void drawStars()
{
    cr16_int listPtrX(r0), listPtrY(r1), x(r2), y(r3), color(r4);
    cr16_int speedRatingPtr(r5), speedRating(r6), tick(r7);
    listPtrX = mem_stars_head;
    listPtrY = (mem_stars_head+1);
    speedRatingPtr = (mem_stars_head+2);
    color = gre;
    memLoad( tick, mem_tick );

    cr16_while( listPtrX < mem_stars_tail );
    {
        load( x, listPtrX );
        load( y, listPtrY );
        load( speedRating, speedRatingPtr );
        drawPixel( x,y, color );

        cr16_if( (tick&speedRating) == 0 );
        {
            ++y;
            cr16_if( y>=30 ); y = 0; cr16_endif();
            store( y, listPtrY );
        }  cr16_endif();

        listPtrX += NEXT_STAR;
        listPtrY += NEXT_STAR;
        speedRatingPtr += NEXT_STAR;
    }  cr16_endwhile();
}  //----------------------------------------------------------------------

void drawParticles()
{
    cr16_int ptrX(r0), ptrY(r1), x(r2), y(r3), color(r4), last(r5);
    ptrX = mem_particles_head;
    ptrY = mem_particles_head + 1;
    last = mem_particles_tail;
color = red;

cr16_while( ptrX < last );
{
    load( x, ptrX );
    load( y, ptrY );
    cr16_if( x > 5 ); // draw particles that are inside play area
    {
        drawPixel( x, y, color );
    }
    cr16_endif();

    ptrX += NEXT_PARTICLE;
    ptrY += NEXT_PARTICLE;
}
 cr16_endwhile();

//-----------------------------------------------
-
void drawPlayer()
{
cr16_int ship_x(r1), ship_y(r2), colGre(r3), colBlu(r4),
memLoad( alive, mem_ship_alive );
memLoad( ship_x, mem_ship_x );
memLoad( ship_y, mem_ship_y );
cr16_if(alive);       // if player is alive, the ship is green-blue
    {
        colGre = gre;
        colBlu = blu;
    }
cr16_else();       // if player is dead, the ship is red
    {
        colGre = red;
        colBlu = red;
    }
cr16_int tick6;     // and exploding
    memLoad( tick, mem_tick );
cr16_if( (tick & 0x000f)==0 );
        createExplosion(ship_x,ship_y);
cr16_endif();

cr16_endif();
drawShip(ship_x,ship_y,colGre,colBlu);
}
//-------------------------------------------------------------------------------
-
void input()
{
    // poll input
    cr16_int this_input(r0), nes_addr(r1), last_input(r2),
    once_input(r3);
    memLoad( this_input, mem_this_input );
    last_input = this_input;
    memSave( last_input, mem_last_input );
    nes_addr = ADDR_NES;
    load( this_input, nes_addr );
    memSave( this_input, mem_this_input );
    once_input = (last_input^0x00ff) & this_input;
    memSave( once_input, mem_once_input );

    // restart game if user holds down SELECT and START
    cr16_if( (this_input&NES_SEL)!=0 & (this_input&NES_START)!=0 );
    {
        loadLabelAddress("StartInvader",r10);
        jCond( UC, r10 );
    }
cr16_endif();

cr16_int alive(r4);
    memLoad( alive, mem_ship_alive );
cr16_if(alive);
    {
        // update ship position
        cr16_int ship_x(r4), ship_y(r5);
        memLoad( ship_x, mem_ship_x );
        memLoad( ship_y, mem_ship_y );
    }
cr16_if( this_input & NES_LEFT ); --ship_x;
cr16 endif();
cr16_if( this_input & NES_RIGHT ); ++ship_x;
cr16 endif();

// ensure ship is in play area
  cr16_if( ship_x < 9 );
    ++ship_x;
  cr16_else();
  cr16_if( ship_x > 54 ); --ship_x;
  cr16 endif();
  cr16 endif();

memSave( ship_x, mem_ship_x );
memSave( ship_y, mem_ship_y );

// shoot bullets
  cr16_if( once_input & NES_A );
  {
    int listPtr(r6), bulletY(r7);
    listPtr = (mem_bullets_head+1);
    // +1 to get the y coord
    cr16_while( listPtr <= mem_bullets_tail ); // find a
    spot in the bullets list, if any
    { load( bulletY, listPtr );
      cr16_if( bulletY == 0 );
      // if find spot, store the bullet's (x,y)
      { store( ship_y, listPtr );
        --listPtr;
        store( ship_x, listPtr );
        listPtr = mem_bullets_tail;
      // end loop
        playSound(SOUND_PLAYER_FIRE);
      }
      cr16 endif();
      listPtr += NEXT_BULLET;
    }
  cr16 endwhile();
  } cr16 endif();
  cr16 endif();       // if (alive)
}

void moveBullets()
{
  cr16_int listPtrY(r0), y(r1);
  listPtrY = (mem_bullets_head+1);
  cr16_while( listPtrY <= mem_bullets_tail );
  { load( y, listPtrY );
    cr16_if( y > 0 );
    {
      --y;
    }
store( y, listPtrY );
} cr16_endif();

listPtrY += NEXT_BULLET;
} cr16 endwhile();

// --------------------------------------------------

void checkIfBulletHitsAlien( cr16_int pBullet, cr16_int pAlien )
{
    cr16_int temp(r4), bulletX(r5), bulletY(r6), alienX(r7),
    alienY(r8), alienHealth(r9);
    temp = pBullet + 1;
    load( bulletX, pBullet );
    load( bulletY, temp );
    temp = pAlien + 1;
    cr16_if( bulletY > 0 );       // if a valid bullet,
    {
        load( alienHealth, pAlien );
        load( alienX, temp );
        ++temp;
        cr16_if( alienHealth > HEALTH_ALIVE );  // and a living alien,
        {
            load( alienY, temp );
            cr16_if( (bulletX >= alienX - 2) & (bulletX <=
            alienX + 2) );
            {
                cr16_if( (bulletY >= alienY - 1) & (bulletY <=
                alienY + 1) );  // collide
                {
                    temp = 0xfffe;
                    alienHealth = (alienHealth & temp) - 1;
                    // then the alien gets hurt
                    bulletY = 0;
                    // and bullet disappears
                    temp = pBullet + 1;
                    store( alienHealth, pAlien );
                    store( bulletY, temp );
                    cr16_if( alienHealth < HEALTH_ALIVE + 2 ); // if the alien dies, create explosion particles
                    {
                        createExplosion( alienX, alienY );
                        playSound(SOUND_ALIEN_DIE);
                        // ** this function invalidates r8,9
                        cr16_int score(r8);
                        memLoad(score, mem_score );
                        score += 4;
                        memSave(score, mem_score );
                    }
                }
            }
        }
    }
    cr16_endif();
} cr16_endif();
void checkIfPlayerHitsAlien( cr16_int pAlien )
{
    cr16_int temp(r4), shipX(r5), shipY(r6), alienX(r7), alienY(r8),
    alienHealth(r9);
    temp = pAlien+1;
    memLoad( shipX, mem_ship_x );
    memLoad( shipY, mem_ship_y );
    load( alienHealth, pAlien );
    load( alienX, temp );
    ++temp;
    cr16_if( alienHealth > HEALTH_ALIVE ); // if a living alien,
    {
        load( alienY, temp );
        cr16_if( (shipX >= alienX-4) & (shipX <= alienX+4) );
        {
            cr16_if( (shipY >= alienY-2) & (shipY <= alienY+2) );
            // hits the player
                temp = 0;
            // player dies
                memSave( temp, mem_ship_alive );
            }
        cr16_endif();
    }
    cr16_endif();
}

void doCollisionDetection()
{
    cr16_int pBullet(r0), pAlien(r1);
    // check each bullet whether it hits aliens
    pBullet = mem_bullets_head;
    cr16_while( pBullet<mem_bullets_tail );
    {
        pAlien = mem_aliens_head;
        cr16_while( pAlien<mem_aliens_tail );
        {
            checkIfBulletHitsAlien( pBullet, pAlien );
            pAlien += NEXT_ALIEN;
        }
        cr16 endwhile();
    }
    pBullet += NEXT_BULLET;
}

137
// check whether aliens collides with player
pAlien = mem_aliens_head;
cr16_while( pAlien < mem_aliens_tail );
{
    checkIfPlayerHitsAlien( pAlien );
    pAlien += NEXT_ALIEN;
}
cr16 endwhile();

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

- void moveAlien( cr16_int pAlien )
{
    cr16_int tick( r2 );
    memLoad( tick, mem_tick );
    cr16_if( ( tick & 0x0f ) == 0x01 );    // move once every 32 ticks
    {
        cr16_int temp( r3 ), y( r4 );
        temp = pAlien + 2;
        load( y, temp );
        ++y;
        store( y, temp );

        cr16_if( y > 30 );    // don't let it go below the screen
        {
            r10 = 0;
            store( r10, pAlien );    // set its health to zero, to make it disappears
        }
    }
    cr16 endif();
}
cr16 endif();

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

- void moveParticles()
{
    cr16_int ptrX( r0 ), ptrY( r1 ), ptrDX( r2 ), ptrDY( r3 ), x( r4 ), y( r5 ),
    dx( r6 ), dy( r7 ), last( r8 );
    ptrX = mem_particles_head;
    ptrY = mem_particles_head + 1;
    ptrDX = mem_particles_head + 2;
    ptrDY = mem_particles_head + 3;
    last = mem_particles_tail;

    cr16_while( ptrX < last );
    {
        load( x, ptrX );
        load( y, ptrY );
        load( dx, ptrDX );
        load( dy, ptrDY );
        cr16_if( x > 5 );    // only mess with particles inside the play area
    }
\[ x' = dx; \]
\[ y' = dy; \]

\texttt{cr16\_if( x>57 \ | \ y<=0 \ | \ y>=29 );} \quad // if particle leaves play area, disable it
\texttt{x = 0;}
\texttt{cr16\_endif();}
\texttt{store( x, ptrX );}
\texttt{store( y, ptrY );}
\texttt{cr16\_endif();}
\texttt{ptrX += NEXT\_PARTICLE;}
\texttt{ptrY += NEXT\_PARTICLE;}
\texttt{ptrDX += NEXT\_PARTICLE;}
\texttt{ptrDY += NEXT\_PARTICLE;}
\texttt{cr16\_endwhile();}

\texttt{//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-}
\texttt{-}
\texttt{void updateAliens()}
\{
\texttt{cr16\_int pAlien(r0);}
\texttt{cr16\_int health(r1);}
\texttt{pAlien = mem\_aliens\_head;}
\texttt{cr16\_while( pAlien < mem\_aliens\_tail );}
\texttt{load( health, pAlien );}
\texttt{cr16\_if( health > HEALTH\_EXIST );}
\texttt{cr16\_if( health > HEALTH\_ALIVE );}
\texttt{moveAlien(pAlien);}
\texttt{cr16\_if( health \& 0x1 );} \quad \texttt{// the}
\texttt{alien is hit by a bullet}
\texttt{--health;} \quad \texttt{//}
\texttt{reduce health, flash color}
\texttt{store( health, pAlien );}
\texttt{cr16\_endif();}
\texttt{// if the alien reaches lower row, score--}
\texttt{cr16\_int score(r8), yPtr(r7), y(r6), alive(r5);}
\texttt{yPtr = pAlien + 2;}
\texttt{load( y, yPtr );}
\texttt{memLoad(score, mem\_score );}
\texttt{memLoad(alive, mem\_ship\_alive );}
\texttt{cr16\_if( y == 29 \ & \ score > 0 \ & \ alive == 1 );}
\texttt{--score;}
\texttt{++y;}

139
memSave(score, mem_score);

    store(y, yPtr);
}

    cr16_endif();
}

    cr16_else();  // the alien is exploding to death
{  //
    --health;  // subtract the explosion counter
    store( health, pAlien );
}
    cr16_endif();
}

    cr16_endif();

    pAlien += NEXT_ALIEN;
}  // end while
}

// findEmptyAlienSlot( cr16_int resultPtr )
{
    cr16_int keepGoing(r6), health(r7);

    resultPtr = mem.aliens_head;
    keepGoing = 1;
    cr16_while( keepGoing );
    {  //
        load( health, resultPtr );
        cr16_if( health <= HEALTH_EXIST );  // this spot is empty. Take it.
        {  //
            keepGoing = 0;
        }  //
    }  //
    cr16_else();  // this spot is occupied
    {  //
        resultPtr += NEXT_ALIEN;
    }
}  //
    cr16_endwhile();
}

// addAlien( cr16_int alienPtr, cr16_int health, int xPos, cr16_int alienType )
{
    cr16_int x(r6), y(r7), xPtr(r8), yPtr(r9), typePtr(r10);
    x = xPos;
    y = -2;
    xPtr = alienPtr;
    yPtr = alienPtr;
typePtr = alienPtr;
xPtr += 1;
yPtr += 2;
typePtr += 3;
store( health, alienPtr );
store( x, xPtr );
store( y, yPtr );
store( alienType, typePtr );
alienPtr += NEXT_ALIEN;
}
//---------------------------------------------------------------
void streamAliens()
{
    cr16_int tick(r0), period(r1), ptr(r2), healthAlien1(r3),
    healthAlien2(r4);
    cr16_int alienType(r5);
    memLoad( tick, mem_tick );

    period = 0x0fff;   // 4096 cycles = 68.27 seconds
    tick = tick & period;

    findEmptyAlienSlot(ptr);
    healthAlien1 = 26;
    healthAlien2 = 34;

    cr16_int elapsedTime(r10);
    memLoad(elapsedTime, mem_tick);
    // increase aliens' HP with time
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( -1, elapsedTime );
    lshi( 1, elapsedTime );
    healthAlien1 += elapsedTime;
    healthAlien2 += elapsedTime;

    cr16_if( tick==60 | tick==1750 | tick==2520 );  // duo - alien1
    {     
        alienType = ALIEN1;
        addAlien( ptr, healthAlien1, 20, alienType );
        addAlien( ptr, healthAlien1, 44, alienType );
    }
    cr16 endif();

    cr16_if( tick==300 | tick==1380 );  // trio partA
    {     
        alienType = ALIEN1;
        addAlien( ptr, healthAlien1, 23, alienType );
    }
findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 41, alienType );
}
cr16_endif();
cr16_if( tick == 360 | tick == 1440 ); /*
 * trio partB
 */
    alienType = ALIEN2;
    addAlien( ptr, healthAlien2, 32, alienType );
} cr16_endif();
cr16_if( tick == 720 | tick == 780 ); /*
 * quadro
 */
    alienType = ALIEN2;
    addAlien( ptr, healthAlien2, 27, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien2, 37, alienType );
} cr16_endif();
cr16_if( tick == 2100 ); /*
 * five aliens
 */
    alienType = ALIEN1;
    addAlien( ptr, healthAlien1, 15, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 23, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 31, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 39, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 47, alienType );
} cr16_endif();
cr16_if( tick == 3180 ); /*
 * six aliens partA
 */
    alienType = ALIEN1;
    addAlien( ptr, healthAlien1, 22, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 32, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien1, 42, alienType );
} cr16_endif();
cr16_if( tick == 3240 ); /*
 * six aliens partB
 */
    alienType = ALIEN2;
    addAlien( ptr, healthAlien2, 22, alienType );
    findEmptyAlienSlot(ptr);
    addAlien( ptr, healthAlien2, 32, alienType );
    findEmptyAlienSlot(ptr);
addAlien( ptr, healthAlien2, 42, alienType );
}
cr16 endif();
//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
-
void addMissile(cr16_int x, cr16_int y )
{
    cr16_int temp1(r5), temp2(r6);
    cr16_int mislPtr(r7), age(r8), mY(r9);

    mislPtr = mem_missiles_head;
    cr16_while( mislPtr < mem_missiles_tail );
    {
        temp1 = mislPtr;
        temp2 = mislPtr;
        temp1 += 1;
        temp2 += 2;
        load( mY, temp1 );
        cr16_if( mY==30 ); // find empty spot
        {
            age = 0;
            store( x, mislPtr );
            store( y, temp1 );
            store( age, temp2 );
            mislPtr = mem_missiles_tail; // break loop
        }
    }
}
cr16 endwhile();

}
//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
-
void aliensShootMissiles()
{
    cr16_int alienPtr(r0), aHealth(r1), aX(r2), aY(r3);
    cr16_int tick(r4), temp1(r5), temp2(r6);

    alienPtr = mem_aliens_head;
    memLoad( tick, mem_tick );
    cr16_while(alienPtr < mem_aliens_tail );
    {
        temp1 = alienPtr;
        temp2 = alienPtr;
        load( aHealth, alienPtr );
        temp1 += 1;
        temp2 += 2;
        cr16_if( aHealth > HEALTH_ALIVE );
        {
            load( aX, temp1 );
            load( aY, temp2 );
            getRand(temp1);
        }
    }
}
cr16_if( (temp1==4) | (temp1==12) );
    // alien fires missile
    playSound(SOUND_PLAYER_FIRE);
    addMissile(aX,aY);
    cr16_endif();
    cr16_endif();
    alienPtr += NEXT_ALIEN;
}
    cr16 endwhile();
//--------
void checkNegativeScore()
{
    cr16_int score(r0), alive(r1);
    memLoad( score, mem_score );
    cr16_if( score == 0 );
        alive = 0;
        memSave( alive, mem_ship_alive );
    cr16_endif();
}
//--------
void updateMissiles()
{
    cr16_int listPtrX(r0), listPtrY(r1), listPtrAge(r2);
    cr16_int x(r3), y(r4), age(r5), tick(r6);
    cr16_int shipX(r7), shipY(r8), temp(r9);
    listPtrX = mem_missiles_head;
    listPtrY = 1;
    listPtrAge = 2;
    listPtrY += listPtrX;
    listPtrAge += listPtrX;
    memLoad( tick, mem_tick );
    memLoad( shipX, mem_ship_x );
    memLoad( shipY, mem_ship_y );

    cr16_while( listPtrX <= mem_missiles_tail );
    {
        load( y, listPtrY );
        load( x, listPtrX );
        load( age, listPtrAge );
        cr16_if( y < 30 ); // if active missile
            cr16_if( (tick&0x3)==0 );
            {
                cr16_if( (x>=shipX-2) & (x<=shipX+2) );
                // if missile hits player
                cr16_if( (y>=shipY-1) & (y<=shipY+1) );
                temp = 0;
                memSave( temp, mem_ship_alive );
                cr16_endif();
            }
        cr16_endif();
    }
move missile
++y;
//

++age;

missile homing

cre16_if( age<8 );

// make

cre16_if( x > shipX+1 ); --x;
cre16 endif();

cre16_if( x < shipX-1 ); ++x;
cre16 endif();

cre16 endif();

store( x, listPtrX );
store( y, listPtrY );
store( age, listPtrAge );
cre16 endif();
cre16 endif();

listPtrX += NEXT_MISSILE;
listPtrY += NEXT_MISSILE;
listPtrAge += NEXT_MISSILE;
cre16 endwhile();

// ---------------------------------------------------------
void updateSound()
{
cre16_int type(r0), remain(r1), soundData(r2), alive(r3);
memLoad( type, mem_sound_type );
memLoad( remain, mem_sound_remaining );
memLoad( alive, mem_ship_alive );

cr16_if( remain > 120 ); remain = 120; cr16 endif();

cr16_if( remain != 0 );
{
    --remain;
soundData = 0x00;
    
    cr16_if( type==SOUND_PLAYER_FIRE ); soundData = 0x0003;
    cr16 endif();
    
    cr16_if( type==SOUND_ALIEN_DIE ); soundData = 0x0004;
    cr16 endif();
    
    cr16_if( type==SOUND_NONE ); soundData = 0x0000;
    cr16 endif();
    
memSave( remain, mem_sound_remaining );

    
    cr16_if( remain==0 );
    type = 0;
memSave( type, mem_sound_type );
    cr16 endif();
}
cre16 else(); // no sound
{  
    soundData = 0x0000;
}

```c
cr16endif();
```

```c

cr16_if( alive==0 );
```

```c

cr16endif();
```

```c
    memSave( soundData, ADDR_SOUND );
}
```

```c
///---------------------------------------------------------------------
void intro()
{
    ifstream f("invader.pix");
    vector<int> x, y;
    float z;

    for( int j=29; j>=0; j-- )
    {
        for( int i=0; i<64; i++ )
        {
            f >> z;
            if( z<0.5 )
            {
                x.push_back(i);
                y.push_back(j);
            }
        }
    }

    r0 = ( ADDR_VGA_PIXELS + y[0]*64 + x[0] );
    r1 = ( ADDR_VGA_PIXELS + y[1]*64 + x[1] );
    r2 = ( ADDR_VGA_PIXELS + y[2]*64 + x[2] );
    r3 = ( ADDR_VGA_PIXELS + y[3]*64 + x[3] );
    r4 = ( ADDR_VGA_PIXELS + y[4]*64 + x[4] );
    r6 = red;
    unsigned int index = 5;
    while( index<x.size() )
    {
        store( r6, r0 );
        store( r6, r1 );
        store( r6, r2 );
        store( r6, r3 );
        store( r6, r4 );
        r0 += (y[index]*64+x[index] - y[index-5]*64-x[index-5]);
        index ++;  if(index>=x.size()) break;
        r1 += (y[index]*64+x[index] - y[index-5]*64-x[index-5]);
        index ++;  if(index>=x.size()) break;
        r2 += (y[index]*64+x[index] - y[index-5]*64-x[index-5]);
        index ++;  if(index>=x.size()) break;
        r3 += (y[index]*64+x[index] - y[index-5]*64-x[index-5]);
        index ++;  if(index>=x.size()) break;
        r4 += (y[index]*64+x[index] - y[index-5]*64-x[index-5]);
        index ++;  if(index>=x.size()) break;
    }
    store( r6, r0 );
    store( r6, r1 );
    store( r6, r2 );
    store( r6, r3 );
```
store( r6, r4 );
flipPage();

r0 = 0;
r1 = ADDR_NES;
cr16_while( r0==0 );
{
    load( r2, r1 );
    r0 = r2 & NES_START;
}
cr16 endwhile();

// print content of reg9 (between 0..999)
// this code uses all registers. save them before you call.
void printReg9()
{
    loadLabelAddress( "skipDrawCode", r1 );
jCond( UC, r1 );
flushPipeline();

    // Code that draw the digit 0,1,2...9
    ifstream f("numbers.pix");
    // read image data of all digits
    vector< int > x[10], y[10];
    float z;
    for( int j=29; j>=0; j-- )
        for( int i=0; i<64; i++ )
        {
            f >> z;
            if( z<0.5 )
            { 
                int which = i/6;
                assert( which>=0 && which<10 );
                // separate them into 10 sets (one for each digit)
                x[which].push_back(i-which*6);
                y[which].push_back(j);
            }
        }
    for( int which=0; which<10; which++ )
    {
        addLabel( "drawDigitLabel" + int2bin(which,16) );
        r0 = ( ADDR_VGA_PIXELS + y[which][0]*64 + x[which][0] ) + r4;
r1 = ( ADDR_VGA_PIXELS + y[which][1]*64 + x[which][1] ) + r4;
r2 = ( ADDR_VGA_PIXELS + y[which][2]*64 + x[which][2] ) + r4;
r3 = blu;
        unsigned int index = 3, size = x[which].size();
        while( index<size )
        { 
            store( r3, r0 );
        }
    }
store( r3, r1 );
store( r3, r2 );
r0 += (y[which][index]*64+x[which][index] -
y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size)
break;
r1 += (y[which][index]*64+x[which][index] -
y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size)
break;
r2 += (y[which][index]*64+x[which][index] -
y[which][index-3]*64-x[which][index-3]); index ++; if(index>=size)
break;
}
store( r3, r0 );
store( r3, r1 );
store( r3, r2 );
jCond(UC,r6); // return
flushPipeline();
}

// r0 = digit to draw, r4=address offset
addLabel("DigitLabel");
cr16_if( r0==0 ); loadLabelAddress( "DigitLabel" +
int2bin(0,16), r1 );
cr16_endif();
cr16_if( r0==1 ); loadLabelAddress( "DigitLabel" +
int2bin(1,16), r1 );
cr16_endif();
cr16_if( r0==2 ); loadLabelAddress( "DigitLabel" +
int2bin(2,16), r1 );
cr16_endif();
cr16_if( r0==3 ); loadLabelAddress( "DigitLabel" +
int2bin(3,16), r1 );
cr16_endif();
cr16_if( r0==4 ); loadLabelAddress( "DigitLabel" +
int2bin(4,16), r1 );
cr16_endif();
cr16_if( r0==5 ); loadLabelAddress( "DigitLabel" +
int2bin(5,16), r1 );
cr16_endif();
cr16_if( r0==6 ); loadLabelAddress( "DigitLabel" +
int2bin(6,16), r1 );
cr16_endif();
cr16_if( r0==7 ); loadLabelAddress( "DigitLabel" +
int2bin(7,16), r1 );
cr16_endif();
cr16_if( r0==8 ); loadLabelAddress( "DigitLabel" +
int2bin(8,16), r1 );
cr16_endif();
cr16_if( r0==9 ); loadLabelAddress( "DigitLabel" +
int2bin(9,16), r1 );
cr16_endif();
jal( r6, r1 ); // call draw
flushPipeline();
jCond(UC,r14); // return
flushPipeline();

// determines what three digits to draw
addLabel("SkipDigitLabel");

r0 = 0; // the 'hundred'
digit

```c

```
r4 = 57 + (64*0);
loadLabelAddress("DrawDigit", r1);
jal(r14,r1);
flushPipeline();

r0 = 0;          // the 'ten'
digit
   cr16_while(r9 >= 10);
      r9 -= 10;
      ++r0;
      flushPipeline();
   cr16_endwhile();
   r4 = 57 + (64*9);
   loadLabelAddress("DrawDigit", r1);
   jal(r14,r1);
   flushPipeline();

   r0 = r9;          //
the 'one' digit
   r4 = 57 + (64*18);
   loadLabelAddress("DrawDigit", r1);
   jal(r14,r1);
   flushPipeline();}
//-------------------------------------------------------------------------------
-
void cr16main()
{
   activateHazardDetectionUnit();
   addLabel("StartInvader");
   initVGA();
   flipPage();

   initGameData();

   intro();

   cr16_while(1);      // Main Game Loop
   {
      updateTick();     // update internal
      states
         input();
         moveBullets();
         moveParticles();
         updateAliens();
         updateMissiles();
         aliensShootMissiles();
         doCollisionDetection();
         streamAliens();
         checkNegativeScore();

         clearBackground();       // draw graphics
         drawBorder();
         drawStars();
         drawPlayer();
         drawAliens();
         drawBullets();
}
drawMissiles();
drawParticles();

memLoad(r9, mem_score);
printReg9();

updateSound();
flipPage();

}  

// End Game Loop
//------------------------------------------