

Computer Design Final Report

**CS/EE 3710
Group 7**

Steve Barrus
Trent Shino
David Reflexia
Rich Wingfield
Owen Xu

Introduction

This document describes the process and results of Group 7's 16-bit design of a 16-computer. The computer instruction set was based on a CR 16 model. Not all of the instructions were implemented (only the required baseline instructions).

The members of Group 7 were: Steve Barrus, David Reflexia, Owen Xu, Trent Shino and Richard Wingfield. The group benefited from a diverse background. This diversity helped bring the project together and allowed for multiple I/O devices to be used.

Group 7 decided early in the project to implement VGA, PS/2 keyboard, and a UART while still adding the parallel port and the seven-segment display listed by the design spec. The addition of the VGA, UART, and PS2 allows for a robust machine that has practical applications.

System Highlights

- 50 MHz CR-16 based RISC CPU
- 16 bit word size
- 16 entry register file
- 16 bit ALU that supports add, subtract, AND, OR and XOR.
- Bi-directional single bit shifter
- 96 KB SDRAM
- 3 KB VGA frame buffer
- 128x96 VGA resolution with 2 bit color
- 19.2 Kbps UART
- PS/2 keyboard interface
- Bi-directional parallel port interface
- Seven segment display output

Design Description

ALU

Our ALU circuit was designed around the ADSU16 macro (a 16-bit adder/subtractor with carry-in, carry-out, and overflow) in the Spartan 2 library. A control signal selects the add/subtract function. The workings of this circuit are described fully in the Xilinx Library Guide. A 16-bit NOR gate was used as a zero detect on the output of this adder and a 2-bit XOR gate with CO and NOTADD_EN as inputs signals a carry-out.

We also implemented the three basic logic functions AND, OR, and XOR in 16 bits each combining multiple 2-bit versions of each into separate units utilizing a 16-bit 4 to 1 multiplexor designed in a similar manner to select through two control signals which function is output.

Another thing we included in the ALU was a shifter which when given a 16-bit number logically shifts it one bit to the left or right as chosen by the sign of the immediate operand of the shift instruction.

Another 16-bit 4 to 1 Mux then selects between the basic logic, adder/subtractor, shifter outputs, and the B input through two control signals. Thus functions our ALU.

ALU_IN_A[15:0]	IN	16 bit data input
ALU_IN_B[15:0]	IN	16 bit data input
BL0	IN	Control signal to choose between logic functions
BL1	IN	Control signal to choose between logic functions
ALU_OUT_S0	IN	Control signal to choose to choose output
ALU_OUT_S1	IN	Control signal to choose to choose output
SUB_ADD/ADD_EN	IN	Selects add or subtract

EN	IN	Enable signal for muxes
CARRY_OUT	OUT	Indicates a carry out
OFL	OUT	Indicates overflow
ZERO	OUT	Indicates zero result

Table 1. Input/Output signals for ALU.

Datapath

One step up in complexity from our ALU is our datapath. We implemented our PC (program counter) with the CB16CLE (16-bit loadable cascadable binary counter) macro in the Spartan 2 library. The function of this circuit is described in detail in the Xilinx Library Guide. Control signals select when to load a value into the PC and when to increment the PC. A 16-bit 2 to 1 Mux is implemented to select the input to be loaded into the PC, be it a register value for a jump-and-link instruction or the ALU output for a conditional branch or jump.

Our IR (instruction register) is just a simple 16-bit register. (FD16CE in the Spartan 2 library) The instruction itself and the enable signal come from the controller.

A 3 input XOR of the carry-out from the ALU and the most significant bit from both ALU inputs creates an input negsignal which is used by the PSR which will be described next.

The PSR (program status register) is simply 4 D-type Flip-Flops that save the carry-out, overflow, and zero outputs from the ALU and the negsignal described earlier and output the five condition codes required by the baseline instruction set.

The register file is another important piece of our datapath. It is described in more detail elsewhere in this report.

For the LUI (load upper immediate) instruction we simply logical left shift 8 bits the immediate value so that it can be stored in the desired register.

For immediate instructions we used buffers and a 1-bit 2 to 1 Mux with one input tied to the sign bit of the immediate value and the other tied to ground. This way we can either sign extend the value for immediate operations such as add and subtract that use two's compliment values or simply zero extend for unsigned numbers.

A 16-bit 2 to 1 Mux selects between the LUI and sign extension circuits as dictated by the control.

Two 16-bit 2 to 1 Muxes are used to select the inputs to the ALU. The first selects between the output value from the PC for jumps and the Dst. register output from the register file. The second chooses between either the immediate value shifted left logically 8 bits or sign-extended or zero extended depending on what was selected previously and the Src. register output from the register file. These selections are made by control signals.

We then have our ALU and its control signals followed by another 16-bit register (FD16CE) like the one used for the IR. This register saves the ALU output to be sent to memory or written to a register or loaded into the PC.

One more 16 bit 2 to 1 Mux is used to select the value written to the register file. It selects between the ALU output and a value from memory.

Register File

The register file was a pretty simple design. We knew we needed 16 16-bit registers, so we partitioned that off into 2 blocks of 16 Deep 8 Wide Dual Port Ram Macros. Using these macros allowed us to quickly design a register file that was fast and reliable. It takes 16-bit data as the input and outputs two sets of 16-bit data separately. This allows us to access two registers at the same time. One block has the top 8-bits of data coming in and 2 sets of 8-bit data going out. The other block has the bottom 8-bits of data coming in and 2 sets of 8-bit data going out.

The register file has several inputs and outputs. It has two 4 bit address lines, a 16-bit data line, and several control lines as input. And it has two 16-bit data lines as output. If the processor were not limited, it would be expandable to a higher number of bits on the address lines, allowing us 32 or maybe 64 registers.

Write	IN	Write enable
Clock	IN	12.5 MHz clock signal from the system
A_Addr_In [3:0]	IN	4-bit address line of Dst Register
D_Addr_In [3:0]	IN	4-bit address line of Src Register
Data_In [15:0]	IN	16-bit data line for writing to a register
SPO_Out [15:0]	OUT	16-bit data line for output of the Dst Register
DPO_Out [15:0]	OUT	16-bit data line for output of the Src Register

Table 2. Input/output signals for the Register File.

The Controller and Decoder

The controller was written to try to minimize the number of states the standard instructions required. The appendix shows the VHDL code used to generate the controller. Figure 3 below shows the states.

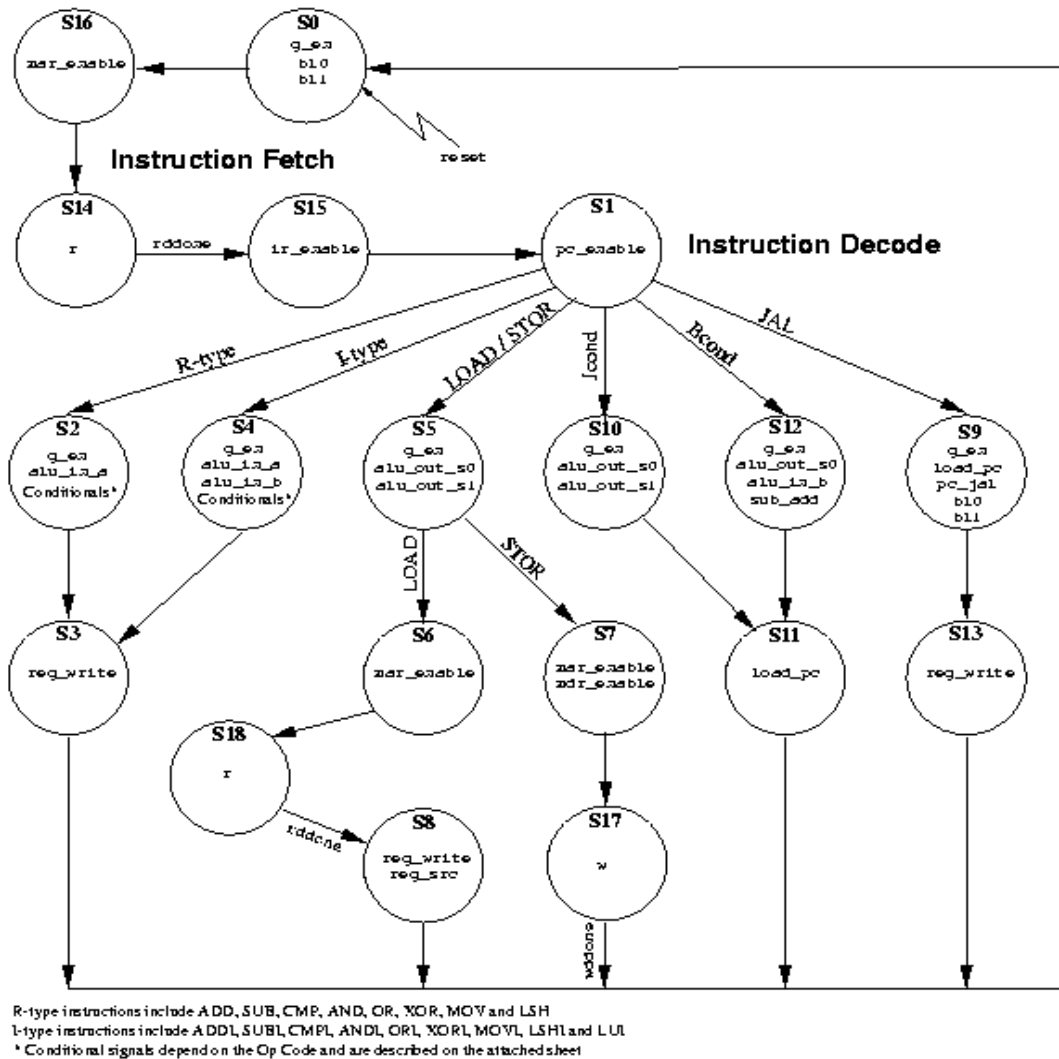


Figure 3. State graph of the controller. (Better picture in the appendix.)

As Figure 3 shows, the first 4 states are used as the instruction fetch. This includes reading the initial/next instruction from memory. It pulls down the initial instruction from memory 0x0000 and starts the looping process. From there, state S1 deciphers the instruction and chooses one of six paths to take.

In the design of the six paths, the group tried to bind the correct instructions to the correct path to minimize both the amount of VHDL code used as well as try to make the controller simpler to modify if needed.

If the instruction is an R-type instruction (ADD, SUB, AND, etc.), the controller goes to state S2 to process the individual R-type instruction. State S3 writes the information to the register. Once the data is written to the register, this (and all routes) returns to state S0 to receive the next command. This starts the looping process from instruction to instruction.

If the instruction is an I-type instruction (ADDi, SUBi, ANDi, etc.) the controller goes to state S4 to handle the direct data read. This instruction set is similar to the R-type instructions with the exception of extracting data directly from the instruction line. Like R-type instructions, this path goes to S3 to write data to the registry and loops back to S0.

If the instruction is a Load or Store, the controller goes to state S5 to determine if it is a Load or a Store. If it is a Load, the state taken is S6 where it sets the address to be read from and waits through S18 until it receives the data. Once received, the controller goes to S8, writes the data to the register, and loops to S0.

If the instruction is a Store instruction, from S5 the state switches to S7 where the address and data bus is populated and WRITE is enabled. The controller waits at state S17 until it receives the DONE command from the memory. Once done, the state goes back to the beginning to get the next instruction.

Because of technical issues related to how the data path was built, it was decided to have the jump instructions each go down their own path for processing. For a JCOND instruction, the controller goes to state S10 to get the data from the instruction. It writes the data to the program counter by setting LOAD_PC high. It then loops back to S0 for the next instruction.

The branch condition moves from state S1 to state S12. It differs from the JCOND command since a subtraction is done with the condition. If the subtraction yields a ZERO, the condition has been met and the state goes to S11. It writes the branch address to the program counter and loops back to S0 to get the next instruction.

The last path is the jump and link (JAL) path. Once the instruction decodes the JAL command, the next state is S9. In S9, the data from the program counter is extracted while the new address is set into the program counter. The next state is S13 where the previous value from the program counter is written to the registers. It then goes back to S0 to fetch the next instruction.

The resulting controller allows for an overall flexible machine. Stalls are handled by the assembler, allowing for a very fast datapath. This places a burden on the programmer while at the same time giving them the freedom to go as fast as they want.

SDRAM Interface

The SDRAM interface is based on the basic layout described in the class notes. It uses a state machine to do basic handshaking with the SDRAM to read and write to it. We chose to implement it in VHDL because of the ease of creating state machines in VHDL. The state machine is implemented using the basic process statement on signals clock and reset, and sets the output signals corresponding to the current state. First it waits for a read or write signal from the controller. Once it gets that, it sends a wr or rd signal to the SDRAM and waits for a response. Once the SDRAM responds with a done signal, the interface returns either a rddone or wrdone signal back to the controller.

Table 4 shows the ports that come out of the SDRAM Interface.

Reset	IN	System reset.
Clock	IN	12.5 MHz clock signal from the system
W	IN	Write signal received from the controller

R	IN	Read signal received from the controller
Done	IN	Done signal received from the SDRAM
WR	OUT	Signal sent to the SDRAM to initiate a write
WRDone	OUT	Signal sent to the controller to notify when a write has completed
RD	OUT	Signal sent to the SDRAM to initiate a read
RDDone	OUT	Signal sent to the controller to notify when a read has completed

Table 4. Input/output signals for the SDRAM Interface.

Memory Mapped I/O

The input and output devices of our CPU can all be accessed via the memory interface. Certain memory addresses are mapped to each of the devices. We a program does a load or a store to the I/O range of memory (0xc000 to 0xffff) it really loads or stores data to or from one of the five I/O devices that we chose to implement. Mapping devices to memory addresses is an easy way to fit a large number of devices into a system. Another way to do this would have been to make separate load and store instructions for the I/O devices, but this would add and even greater complexity to the controller and the instruction decoder, but since using memory mapping was required by this course, this really does not matter. In order to reduce the complexity of our processor, we chose not to implement interrupts. Interrupts would have given some advantages, but we decided nothing that we wanted to require the added complexity.

The memory mapped I/O portion of our processor design consists of two main parts: the address decoder and the device interfaces. These two components would together provide an efficient way to communicate to the various input and output devices that we chose to work with. The address decoder looks at the address given to any memory access and sends out enable signals to the various devices that are mapped in to memory. It has a total of six enable signals, one global enable and one for each of the devices. The global signal is turned on whenever a memory access occurs in the upper quadrant of RAM (0xc000 to 0xffff). Each of the five devices that we implemented has an address associated with them that is within this address range. The following table shows the available devices and the addresses associated with each.

Address	Device	Attributes	Comment
0xc000	Seven Segment Display	Read/Write	Lower 7 bits tied to segments
0xc100	Parallel Port	Read/Write	Read 8 bits and write 3 bits
0xc200	UART Send/Transmit	Read/Write	Read from the receive register and write to the transmit register
0xc300	UART Status/Control	Read/Write	Read from the status register and write to the control register
0xc400	PS/2 Keyboard	Read Only	Read last keyboard scancode
0xd000 – 0xdbff	VGA	Write Only	Write to the VGA frame buffer

Table 5. Memory mapped I/O addresses.

One of the disadvantages of our design is that our I/O reads and writes are slow. We chose let all of I/O writes to be written to the SDRAM as well. This made our writes a lot slower than they need to be. The advantage of this design is that it made all writes take the same amount of time regardless of where the write was taking place. This also made debugging the I/O subsystems easier because anything written to an I/O device also was written to SDRAM, making a memory dump very useful. Without this finding out what was being written to the device would be impossible without directly checking that device. The reads were also slower, but that provides no advantage except consistency. All reads and writes always take the same number of clock cycles.

I/O Subsystems

The Seven Segment Display

The interface for the seven-segment display is relatively simple. It has a single eight-bit register that is tied directly to the individual segments of the display. The first bit is connected to segment 1; the second bit is connected to segment two, and so on. Since there are only seven segments the eighth bit is ignored. When a program does a store to address 0xc000, the lower byte of the word gets stored in the register (and therefore gets displayed). When a program does a load from that same address, it will read back the value that is in that same register.

The Parallel Port Interface

The parallel port interface (PPI) is similar to the interface for the seven-segment display, except it has two registers. The PPI has an eight-bit register for receiving data and a three-bit register for sending. The receive register is tied directly to the eight data bits of the parallel port and the send register is tied to the three status lines. The three to eight ratio makes communication a little lopsided, but this was the specification that was assigned to us. When a program does a load instruction at address 0xc100, the data from the receive register is read in. The send register can only be written to by the parallel port. Actually, it only gets written to by the device that is connected to the parallel port. On the other side, the send register is write only. When a program does a store at address 0xc100, the send register gets the lower three bits of the data that is being written.

The UART

A UART is a universal asynchronous receiver and transmitter. It is a circuit that sends and receives data serially (one bit at a time) as specified by the EIA-232 standard. Our UART consists of two main parts: a sender and a receiver. Both the sender and the receiver have eight bit registers attached to them that are mapped into memory at 0xc200. When a program does a read from this address, it will get the data from the receiver register (or buffer). When a store is performed at this same address, the data will be written to the transmit register. Simple loading and storing to and from this address is not enough to make the UART work. There is also a status/control register that needs to be used. The control/status register is at the address 0xc300 and must be watched carefully in order to make a successful communication. Our UART only allows for one byte to be in the send and receive registers at a time, so there is no room for overflow. This means that if you receive one byte of data, you must quickly read it from the receive register before you can get another one.

If a program is trying to receive data from the UART, it should watch the status register until the first bit goes high (is a 1). This bit tells the program that a byte was received and is waiting to be read from the receive register. The program should then read this value by doing a load from 0xc200. Then the program should signal that it got the data and that it is OK for the receiver to get more data. It can do this by writing a 1 to the control register (0xc300). Finally the program should wait for the first bit of the status register to go low. This shows that the receiver has acknowledged that the program has the data and it will continue to receive more data. The program can then repeat this process to receive more data.

To send data with the UART, the program must first write the data that it wishes to send to the transmit register at 0xc200. Once that register has been written to, the program needs to raise the transmit request bit of the control register, which is the second bit. Once this bit is raised, the program needs to watch the second bit of the status register for the acknowledgement that the data has been sent. When the transmission has been acknowledged, you need to lower the transmit request bit and wait for the transmission acknowledgement to go low again. This process can then be repeated to send multiple bytes of data.

PS/2 Subsystem

The group decided to implement a PS/2 port interface into the computer. It was decided that a keyboard would be a nice addition to the computer. The PS/2 core would write to a flip-flop one bit at a

time. Once the keyboard loads the flip-flop with the 8-bit character, the register would write that data to memory address 0xc400 where it can be accessed by any application in memory.

Table 6 below shows the connection information to and from the PS/2 core code.

rst	OUT	System Reset
oeb	OUT	RAM Output Enable
kb_data	IN	Serial Data from the keyboard
db	OUT	Bargraph LED (Unused)
rsb	OUT	Output of keyboard data

Table 6. PS/2 Interface

Anyone who wishes to write assembly code will need to code for the keyboard commands returned from the keyboard (stored in memory address 0xc400. Table 7 shows a breakdown of those codes.

KEY	MAKE	BREAK		KEY	MAKE	BREAK		KEY	MAKE	BREAK
A	1C	F0,1C		9	46	F0,46		[54	F0,54
B	32	F0,32		`	0E	F0,0E		INSERT	E0,70	E0,F0,70
C	21	F0,21		-	4E	F0,4E		HOME	E0,6C	E0,F0,6C
D	23	F0,23		=	55	F0,55		PG UP	E0,7D	E0,F0,7D
E	24	F0,24		\	5D	F0,5D		DELETE	E0,71	E0,F0,71
F	2B	F0,2B		BKSP	66	F0,66		END	E0,69	E0,F0,69
G	34	F0,34		SPACE	29	F0,29		PG DN	E0,7A	E0,F0,7A
H	33	F0,33		TAB	0D	F0,0D		U ARROW	E0,75	E0,F0,75
I	43	F0,43		CAPS	58	F0,58		L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B		L SHFT	12	F0,12		D ARROW	E0,72	E0,F0,72
K	42	F0,42		L CTRL	14	F0,14		R ARROW	E0,74	E0,F0,74
L	4B	F0,4B		L GUI	E0,1F	E0,F0,1F		NUM	77	F0,77
M	3A	F0,3A		L ALT	11	F0,11		KP /	E0,4A	E0,F0,4A
N	31	F0,31		R SHFT	59	F0,59		KP *	7C	F0,7C
O	44	F0,44		R CTRL	E0,14	E0,F0,14		KP -	7B	F0,7B
P	4D	F0,4D		R GUI	E0,27	E0,F0,27		KP +	79	F0,79
Q	15	F0,15		R ALT	E0,11	E0,F0,11		KP EN	E0,5A	E0,F0,5A

R	2D	F0,2D		APPS	E0,2F	E0,F0,2F		KP .	71	F0,71
S	1B	F0,1B		ENTER	5A	F0,5A		KP 0	70	F0,70
T	2C	F0,2C		ESC	76	F0,76		KP 1	69	F0,69
U	3C	F0,3C		F1	5	F0,05		KP 2	72	F0,72
V	2A	F0,2A		F2	6	F0,06		KP 3	7A	F0,7A
W	1D	F0,1D		F3	4	F0,04		KP 4	6B	F0,6B
X	22	F0,22		F4	0C	F0,0C		KP 5	73	F0,73
Y	35	F0,35		F5	3	F0,03		KP 6	74	F0,74
Z	1A	F0,1A		F6	0B	F0,0B		KP 7	6C	F0,6C
0	45	F0,45		F7	83	F0,83		KP 8	75	F0,75
1	16	F0,16		F8	0A	F0,0A		KP 9	7D	F0,7D
2	1E	F0,1E		F9	1	F0,01]	5B	F0,5B
3	26	F0,26		F10	9	F0,09		;	4C	F0,4C
4	25	F0,25		F11	78	F0,78		'	52	F0,52
5	2E	F0,2E		F12	7	F0,07		,	41	F0,41
6	36	F0,36		PRNT	E0,12,	E0,F0,		.	49	F0,49
				SCRN	E0,7C	7C,E0,				
						F0,12				
7	3D	F0,3D		SCROL L	7E	F0,7E		/	4A	F0,4A
8	3E	F0,3E		PAUSE	E1,14,7	-NONE-				
					7,					
					E1,F0,1					
					4,					
					F0,77					

Table 7. Scan codes returned from a PS/2 keyboard.

The design of the PS/2 interface is limited in that the buffer for the characters is one. This should not be a problem for the programmer since if they need to store more than one character, they can store it in another part of memory. The programmer will also need to write a code to “empty” the contents of the memory address that the keyboard is writing to so that if the operator types the same key twice, the program knows it.

This design of the PS/2 gives the programmer and assembler ultimate control over how they want to use the codes returned from the PS/2. Both codes (make and break signals) are trappable by the assembler and programmer if needed. The PS/2 clock limits the speed of the PS/2, but the update of the register is running at the speed of the system clock. Thus, the register is updated very quickly.

VGA Subsystem

The group decided early on to incorporate the use of the VGA capabilities of the XS-50. The class web page provided some information concerning the VGA core. The appendix shows the VHDL code for the final version of the core. Once the core was written, the decision on how to hook it into the current processor was discussed. The group decided to use the block ram provided on-board to allow for speed and ease. Using the block-ram did cause some problems.

The block ram allowed for easy and quick use for the video buffer. However, the block ram was small. This meant that the colors used had to be small. Per the core spec, the VGA uses a 2-bit color scheme. This allowed for red, blue, and green. There is also white and black used for blanking.

The size of the block ram also meant that the size of the screen had to be reduced. The resolution that the block ram could hold was 128x96. This proved to be too small to be useable, so we ended up counting a 2x2-block set per pixel. This allowed for a larger, easier to use screen, but it was a bit blocky.

Per the VGA spec, the clock going into the Core was reduced. The system works off of a 50 MHz clock signal. This signal is much too fast for the VGA Core. So the clock going to the video was divided by four using a counter. This provided a 12.5 MHz clock signal when the spec called for 12 MHz.

Table 8 below shows a breakdown of the VGA Core code.

A	Horizontal pixel counter.
B	Vertical line counter.
C	Operation of the horizontal sync pulse generator.
D	Operation of the vertical syn pulse generator.
E	Describes the computation of the combinatorial blanking signal.
F	Describes the operation of the pipelined video-blanking signal.
G	RAM is disabled. This is currently not used with this processor.
H	Address in RAM where the next four pixels are stored.
I	Describes the operation of the register that holds the byte of the pixel data read from RAM.
J	This process describes the process by which the current active pixel is mapped into the six bits that drive the red, green, and blue guns.

Table 8. VGA Core code breakdown.

Table 9 shows the ports that come out of the video Core.

Reset	IN	System reset.
Clock	IN	12.5 MHz clock signal from the system
Hsyncb	OUT	Horizontal sync sent to the monitor
Vsyncb	OUT	Vertical sync sent to the monitor
RGB	OUT	6-bit red, green, blue signal sent to the monitor for the pixel
Addr	OUT	15-bit address of next 4 pixel data in block RAM.
Data	IN	8-bit data from block RAM. This represents four, 2-bit pixels.
CSB	OUT	Not used. RAM Chip Enable.
OEB	OUT	Not used. RAM output enable.
WEB	OUT	Not used. RAM write enable.

Table 9. Input/output signals for the VGA core.

VGA Block Ram

The block ram takes in a 12-bit address line, an 8-bit data line, a couple of control signals, and outputs an 8-bit data line. The first nine address bits are passed to the block ram macro, RAMB4_8. The

top 3 address bits are passed to a decoder, which then determines which of the six block ram macros to write to/ read from. Then the correct data is extracted to the output line with an 8 bit, 8 to 1 mux.

The block ram can operate in two modes. One: It takes the address and data from the processor via a load command and writes to the specified location in the block ram. Two: It takes the address from the VGA core and outputs the appropriate data to the VGA core. The mode is determined by the address on the load command. If the process does not want to write to the block ram, then the VGA core is allowed to read from it. If the processor IS writing to the block ram, then the VGA core is NOT allowed to read from it. This does have a few downfalls. In our animation program, there are random glitches on the screen. This is caused when the VGA core wants to read the ram, but we are writing to it.

We tried to find a solution to this, but ultimately ran out of time. The glitches are pretty minimal so it was not a big issue.

Clear	IN	System reset.
Clock	IN	12.5 MHz clock signal from the system
Addr_In9	IN	One of the top 3 addr bits to determine which Block ram to access
Addr_In10	IN	One of the top 3 addr bits to determine which Block ram to access
Addr_In11	IN	One of the top 3 addr bits to determine which Block ram to access
Addr_In [8:0]	IN	9-bit address for writing to/reading from block ram
Data_In [7:0]	IN	8-bit data from processor. This represents four, 2-bit pixels
Data_Out [7:0]	OUT	8-bit data to VGA Core. This represents four, 2-bit pixels
En	In	Enable
Write_En	In	Write Enable

Table 10. Input/output signals for the Block Ram.

Testing

Effective and comprehensive testing may be the most important aspect of the entire project. Throughout the construction of our machine – from the basic logic units to the SDRAM interface to the I/O extensions – we used incremental testing before stepping up to the next hierarchical function. For as complex as the final machine may be, it is important to remember that it is still a collection of many simpler systems.

For the majority of the semester, our only means of feedback on our system was running simulations through the Fusion simulator in Powerview. Our first major simulations took place after we completed our ALU. This involved hard-coding individual instructions to match the instruction format in the EECS 427 data sheet, then hand-setting all of the mux settings and enable inputs. Once we had all these settings loaded as desired, we would run the simulator and check the incoming result – i.e. check the destination register for an arithmetic operation. We tested all baseline instructions with at least two or three random values, including negatives for arithmetic instructions. Figure 11 below shows a sample of the simulating procedure from this stage of development.

```
echo ADD $0, $1
a ir 0001\h
a alu_in_a 1
a alu_in_b 0
a alu_out_s0 1
a alu_out_s1 0
a reg_source 0
a sub_add 1
c
```

Figure 11. simulation code for an ADD instruction assuming registers 0 and 1 have been pre-loaded with values. Note the manual assignments of various mux settings

```

a reg_write 1
c
a ir 0100\h
a reg_write 0
c
check dout A\h

```

The next step of simulation was to perform reads and writes to the SDRAM and memory-mapped I/O. Using a handshake controller for the SDRAM interface, we could store some data by hard-coding an instruction and then also load the data back into a register. It turned out that we should allow around 9 cycles to perform a store and 13 cycles for a load for the SDRAM. For the memory-mapped I/O (7-segment LED and parallel port), we drew schematics that would identify the quadrant of memory for the I/O devices and from those, we could simulate whether the devices would work in hardware.

Simulation was made a great deal more efficient once we implemented our controller circuit. Using Powerview's loadm and dumpm commands, we were able to form simulated memory that verified that our interface indeed was working correctly. Figure XX to the right shows what kind of simulation was available after the controller was working correctly.

```

bitrange 15:0 | define memory width
default value 00 | defines value in unused addresses
default radix hex
adrs 00 | address associated with next value
d002 | MOVI $0, 2
5001 | ADDI $0, 1
d110 | MOVI $1, 30
d230 | MOVI $2, 30
0190 | SUB $1, $0
0051 | ADD $0, $1
9005 | SUBI $0, 5
8001 | LSHI $0, 1
4042 | STOR $0, $2
5201 | ADDI $2, 1
4142 | STOR $0, $2
0021 | OR $0, $1
5201 | ADDI $2, 1
4042 | STOR $0, $2
d300 | MOVI $3, 0
4103 | LOAD $1, $0
5201 | ADDI $2, 1
4142 | STOR $1, $2

```

Figure 12. Example of loadm format using the controller to decode entire instructions

After mapping our design to a .bit file, and having the assembler complete, the testing consisted of running small programs that make use of the baseline instruction set such as in-out.s seen below.

<pre> // in-out.s .org 0x3000 // 7-seg table .word 0x0077 // 0 .word 0x0012 // 1 .word 0x005d // 2 .word 0x005b // 3 .word 0x003a // 4 .word 0x006b // 5 .word 0x006f // 6 .word 0x0052 // 7 .word 0x007f // 8 .word 0x007a // 9 </pre>	<pre> .org 0x0000 main: lui 0xc0 r7 //7-seg display lui 0xc1 r8 //PPI input lui 0x30 r4 //table of digits loop: movi 0 r0 add r4 r0 load r1 r0 stor r1 r7 load r0 r8 //get input from // PPI </pre>
--	---

<code>.word 0x007e // A</code>	<code>andi 0x3f r0 //only keep the</code>
<code>.word 0x002f // b</code>	<code>// lower 5 bits</code>
<code>.word 0x0065 // C</code>	<code>cmpi 0x10 r0 //r0 < 0x10</code>
<code>.word 0x001f // d</code>	<code>bls loop</code>
<code>.word 0x006d // E</code>	<code>movi 0x10 r0</code>
<code>.word 0x006c // F</code>	<code>buc loop</code>
<code>.word 0x0037 // U</code>	

Figure 13. Simple assembly program used to test baseline instructions as well as I/O from the Parallel Port and the 7-segment LED display

At this point, the assembler was still working out some bugs as well as the circuit. The final tests run were those using various I/O devices, which would test modifications we were constantly making to the machine. These tests were a cycle that started with new schematics to debug / simulate and then went on to running test programs, testing the memory-mapped I/O.

Testing small programs and lower levels of the design hierarchy led the way for our team to make more complex designs that could make use of the I/O interfaces we constructed.

Assembler

The assembler was programmed using the Java language. Java is a great high-level, object-oriented language and it allowed us to write a working assembler in a short amount of time.

We worked on the assembler at the same time as the controller. Splitting up the work in this parallel way allowed us to run programs as soon as the control was finished.

The assembler reads the input file one line at a time. It then parses that line into an Instruction class with all the different parts separated. It will then call the various output functions for the different formats. It will output the following formats: lst, log, dat, sym, and hex. We chose to use the Intel Hex format mainly because of the checksum. We felt it was a great way to check for errors with the assembler.

We programmed the assembler to recognize labels. We felt this would really help in writing complex assembly programs. The assembler will handle all of the base line instructions and a few of the dot instructions. It will recognize `.org`, `.word`, `.end` and `.wordlabel`. We found the last one useful because of the jump and link instruction. Since that instruction jumps to a register, we needed a register with the address of the label we wanted to jump to. `.wordlabel <label>` will find the address of the label, and input it as a word. Then the register can pull it out of a `wordlabel` table when needed.

The assembler is also capable of handling different number formats. It will recognize normal decimal numbers, hex numbers if they have an '0x' in front, and binary numbers if they have a 'b' in front. This caused a few problems since we had to implement various functions to convert all the formats into decimal so we can use them.

This assembler has good error handling, but is still vulnerable to certain kinds of syntax errors. It outputs all the different kinds of errors to the log file and we used that many times to fix our bugged programs. The problem the assembler has is mostly with lines that are empty but still has a 'tab' in it. That is the only time when it will cause a run-time error. Most other syntax errors, it will finish the program and output the error message into the log file. Another issue with it is that the branches are limited to 256 lines of instruction. This is more an issue with the limit of the 16-bit processor.

Use

The assembler was made available to everyone in the group via the web address: <http://www.cs.utah.edu/~oxu/cs3710/Assembler.zip> It is fairly well documented and all were welcome to change it or fix it as they saw fit. The following instructions were given to the group on how to implement the assembler:

1. Install a Java Development Kit. You can get it off Sun's website <http://java.sun.com>.
2. Setup "Path" and "Classpath" in your environment variables. Your path needs to include the path to the jdk/bin. Ie. "Path = c:\jdk1.3.1\bin;%Path%" Your classpath needs to include the class folder of the assembler. Ie. "Classpath = C:\JavaAssembler\classes;%Classpath%"

3. Move your assembly file into the same folder as your classes. Ie. C:\JavaAssembler\classes
4. Open up a command window and type the following: “java Assembler <file.s> <outputfilename (no extensions)> “Assembler” is case sensitive.
For example, if I had an assembly program named “testcode.s” and I wanted the output to be testcode.hex, .lst, .sym For example, type “java Assembler testcode.s testcode”
5. If there is a run-time error, go back through your program and make sure all the blank lines are truly blank. Also, limit the end of your program to only include one blank line.
6. One of the output formats is named .MCS, but it is really an Intel Hex format file.

With the instructions and the completed assembler, the group had a working, simple to use assembler to compile anything needed to run.

Demo Application

Description

The application that we chose to demo was fairly simple in concept. We designed a program that could display animations using our VGA interface. The animations were all constructed before the demonstration and were all displayed by our animation program. We chose this program because it was easier enough to implement in the time that we had and it did an excellent job of showing of the capabilities of the most (visually) interesting component of our system; the VGA interface.

The program worked by loading one frame at a time from the SDRAM to the VGA frame buffer. After a frame was load, the program would invoke a delay function that we wrote. The time of the delay was determined by the data that it read in from the PS/2 keyboard. That way the speed of the animation could be changed by a single keypress. This was useful and it also showed of the capability of our PS/2 keyboard interface. After the delay, the program would then continue to the next frame and continue to the end of the animation. At that point, it would cycle back to the beginning.

Application Interface

This program set us apart from the other groups because none of the other groups did animations. One drawback was that it didn't show off all of the extensions that we added to our project. It really only showed the VGA and PS/2 port, leaving the parallel port interface, the seven segment display and the UART out in the cold. We left out these other extensions because we figured that no one wanted to see it unless it was flashy, like the VGA. We did come up with a demo program for the other extension, but we never showed it off. (Mostly because it was boring.) It can be seen in the appendix (uart2.s).

Using this program was fairly simple. You simply need to use the Xsload tool to the program and one of the animation hex files into memory and our system took care of the rest. Once the program was running, you could press any key on the PS/2 keyboard to change the speed of the animation. Due to memory constraints, there was no way to change the animation while the program was running. Only one animation would fit into RAM at a time. You have to use the Xsload tool to get a different animation. We had seven animations prepared for the Demo Day: a spinning globe, a man running in a box, a running puma, two stick figures fighting, a pencil that drew a face, a water drop splashing, and a galloping horse. We would have prepared more, but they were time consuming to make.

Assembly Listing

```
//
// animate_ps2.s
//
.org 0x0000
        lui 0x01 r6           // animation location
        load r7 r6           // x res
        addi 1 r6
        load r8 r6           // y res
        addi 1 r6
        load r9 r6           // x offset
```

```

        addi 1 r6
        load r10 r6          // y offset
        addi 1 r6
        load r11 r6         // number of frames
        addi 1 r6
        load r12 r6         // delay
        addi 3 r6
        lui 0xd0 r5         // VGA I/O address
        lui 0xdb r0         // end of VGA I/O space
        ori 0xff r0
        movi 0xff r2        // four white pixels
blank:   stor r2 r5          // blank the screen
        addi 1 r5
        cmp r5 r0
        bls blank
cycle:   mov r6 r4
        movi 0 r3
frame:   lui 0xd0 r5
        movi 0 r1
        add r10 r5          // y offset
ver:     movi 0 r0
        add r9 r5           // x/4 offset
hor:     load r2 r4
        stor r2 r5
        addi 1 r4
        addi 1 r5
        addi 1 r0
        cmp r7 r0           // x resolution
        bhi hor
        add r9 r5           // x/4 offset
        addi 1 r1
        cmp r8 r1           // y resolution
        bhi ver
        movi 0x80 r14
        jal r15 r14
        addi 1 r3
        cmp r11 r3         // number of frames
        bhi frame
        buc cycle

.org 0x0100                    //delay function
        lui 0xc4 r5
        load r1 r5
        andi 0x0f r1
        mov r1 r13
delay1:  lui 0xff r14
delay2:  subi 1 r14
        cmpi 0 r14
        bne delay2
        cmpi 0 r13
        subi 1 r13
        bne delay1
        juc r15

```

Figure 14. Demo application assembly code.

Conclusion

This project helped “de-mystify” the computer design building process for the Group 7 members. Not only did the design of the CPU turn out well, but also the add-on peripherals worked! The group spent a lot of time together working as a team and coordinated the building of the project.

Most of the design process was straight forward with the exception of the last minute changes to the PS/2 subsystem. This eleventh-hour addition made it difficult to fully implement into a working demo, but the group pulled together and had something to show that the PS/2 design worked. There was no busy work in this project since the group decided on what it wanted to implement.

The use of the XESS boards made the entire project possible. The lack of wire wrapping made it possible for the group to implement all of the peripherals that the group desired to do. The use of these boards allowed us to spend more time learning how to design and implement than wasting time wire wrapping boards.