CPSC / ECE 3710, Sep 7
Assembler design
Memory Subsystem Design

Microprocessor Design Lab
Ganesh Gopalakrishnan

You are in for a treat !!

- You are gonna build an actual processor!!
  - Fab in silicon no big deal these days, if you want only a medium-fast processor
- You are gonna write an assembler and simulator!
- You are gonna learn VHDL and high level synthesis
- Once the basic CPU works, you can add
  - Interrupts
  - Pipelining
- You can EVEN put TWO cpus onto one xilinx chip, make them feed off the tri-ported memory
  - One CPU can be a graphics co-processor
- Put these on your resume and watch heads turn !!!!!!!
  - No kidding…
- Highly recommend applying to grad schools
  - If you didn’t know what it entails, ask us (and attend CE seminars)
  - I’m talking about research overseas on 9/19
  - Myers talked about Grad School – so might Regehr in his talk 9/12

Assembly Language Design

- An assembler allows one to begin using a CPU comfortably
  - Don’t have to deal with the CPU at the bit level
  - Yet one does not also have to write a C compiler (or other compilers)
    - Too much work!
    - Besides, one needs an assembler before one writes a C compiler
      - This is because compilers also generate assembly code !!!

Assembler Syntax, including Instructions and Directives

- After decades, people have figured out the essentials of a simple assembler
  - Instructions -- add, movi, etc
  - Directives -- .org, .word, .equ, .move_word, .end
  - Most directives control HOW an assembler generates code
  - Some directives generate multiple machine instructions (are really macros)
  - A two-pass algorithm is easiest to use

Example Assembly Program

```
.org 0xBFFF    ; Defines start of code / data block
   L1: .word 0xA0B0   ; The word A0B0 sits here
   L2: .word 0xC0D0   ; Here sits the data word C0D0
   L3: .word 0x0000   ; and finally 0000

      .org 0x0080    ; Now move to a different origin
   L7: .equ *
      movi L7, r1    ; Here, we move L7 (a byte) into r1

      .org 0x0800    ; Move to another origin
   L4: .move_word L2, r2    ; Move word L2 into r2 through a series of actual
                  ; machine instructions
   L8: add  r1, r2    ; add r1 and r2
      bcs  L6        ; Branch on carry set to L6 (forward branch)
      movi l3, r3    ; move immediate 13 into r3
      movi -l3, r3   ; Now move negative 13 (in 2's complement)
      movi 0xF3, r3  ; This also moves negative 13 into r3 (2's complement)

   L5: .equ L4       ; equate L5 to L4
      stor r2, r3    ; Store r2 into mem[r3]
      lshi -3, r4    ; This is essentially right-shift of 3 places
      lshi 0x1F, r4  ; This would be directly interpreted as rsh of 15 places

   L6: .equ *        ; Let L6 be the same as the present PC value
      beq  L4        ; Branch, generating a negative offset
      .end           ; This serves as a clean end-of-program marker
```

Example Listing Output

```
                 .org 0xBFFF
BFFF: A0B0   L1: .word 0xA0B0
C000: C0D0   L2: .word 0xC0D0
C001: 0000   L3: .word 0x0000

             .org 0x0080
          L7: .equ *
0080: B180       movi L7, r1

             .org 0x0800
0800: F2C0   L4: .move_word L2, r2 -- expands into LUI, ORI
0801: 2200
0802: 0251   L8: add  r1, r2
0803: C206       bcs  L6         -- displacement is +6
0804: D30D       movi l3, r3
0805: D3F3       movi -l3, r3
0806: D3F3       movi 0xF3, r3

          L5: .equ L4
0807: 4243       stor r2, r3
0808: 8413       lshi -3, r4
0809: 841F       lshi 0x1F, r4

          L6: .equ *
080A: C0F6       beq  L4       -- branch offset is -10, which is F6 in 2's-C
             .end
```

Designing Assemblers

- Use a two-pass algorithm
  - Pass-1 : forms symbol table of labels and values
  - Pass-2 : generates code
- Most assemblers can do it in 2 passes
- Some assemblers can't (e.g., MC 6800)
  - jmp Lab
  - …
  - Lab: <some instruction>
    - Can generate short jmp instruction if jumping only +/- 128 locations from current PC
    - But that then affects the address of Lab
    - So seeing Lab, we can't say its value; its value is CONDITIONAL on whether a short or long jmp instruction was selected
    - Now imagine a "rats nest" of such jumps
      - In this case one has to formulate all the constraints that apply, and apply a powerful CONSTRAINT SOLVER

Summary of assembler instructions

- Handles all opcodes
- Official syntax is here
  - <label> <opcode> <operands> ; comments
  - <label> = BLOCK012:
    - See BNF grammar for ALL details
  - <opcode> = all your opcodes
  - <operands> = comma separated if more than one
  - See example program for details

Summary of assembler directives

- .org address
  - sets address at which to begin assembly (till next .org)
- .word <byte1><byte2>
  - Patches memory with these bytes (generates these bytes in the HEX load file)
- <label> .equ <value>
  - Sets the label to stand for a value
- .move_word <label>, <regno>
  - Generates two instructions – an LUI and an ORI – to move the 2-byte value of <label> into <regno>
  - We don't do LUI and ADD because then the flags would be affected
- Note: I've specified opcodes to be lowercase and labels to be upper case. If you handle OTHER cases also, that's just fine
- <label> .equ * -- sets <label> to equal the current PC
  - Uses : If you have patched .word locations to be the interrupt vector, then the <label> .equ * will allow that <label> to point to the interrupt vector
  - Some assemblers allow arithmetic
    - e.g., * + 2   or Label – 4   etc
    - This is optional in your assembler
- All values must fit within allowed range – e.g. if a byte is expected, but the label provided exceeds this value, an error should be generated
  - Initial assembler versions may allow these error checks to be left out ('user beware' mode of using it …)

Other Instruction Details

- Notice that we have word addresses
  - So each address refers to the address of a 16-bit word
  - Misaligned accesses not permitted
    - Can't manufacture a word out of the lower byte of one word and the upper byte of another
  - Loader: deals with byte addresses (each next address is that of the next byte)
  - This is to be watched for in the hex loader files
- I've decided to use a sign-magnitude format for LSH and LSHI
- All negative values to be provided in one of two ways
  - in decimal, as in -13
  - in hex, in 2's complement, as in 0xF3

Loader Details

- Generate MCS records
  - Byte_count   Start_Addr   Type_of_record   Bytes   Checksum
  - Example: :10_0170_00_70_71_..._7E_7F_07
  - Checksum : 2's compl of sum of all bytes
    - 10 + 01 + 70 + 71 + … + 7E + 7F ➔ drop carries → 2's compl
  - Type_of_Record could be an address extension record
    - :02_0000_04_0001_F9
      - 2 bytes of data == 00 and 01
      - Now for every 00 type record, OR with 010000
      - Example: If 10_0170_00_70_... follows the above addr extn record then the address of loading = 10170

Simulator Details and MISC. advice

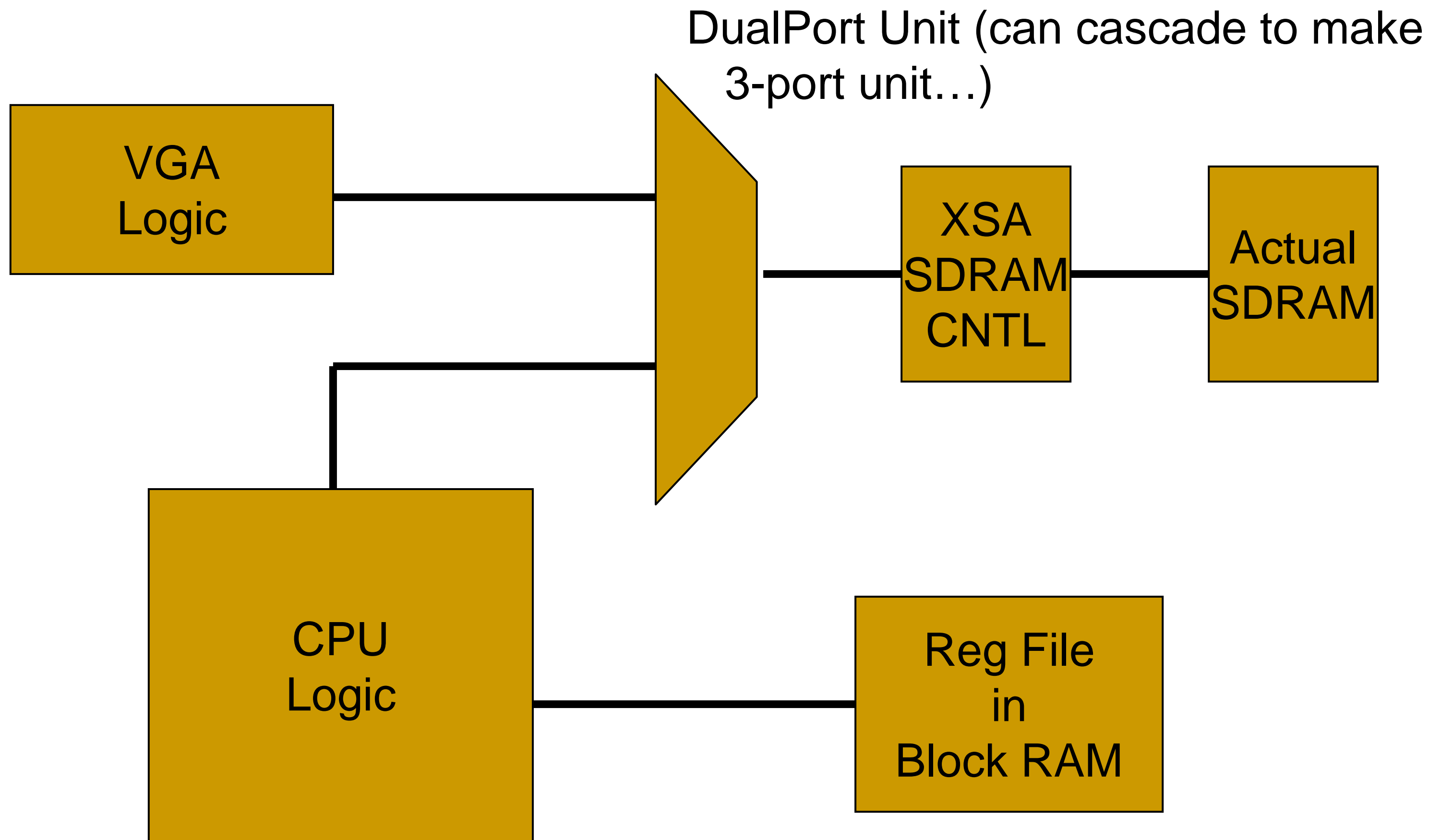
- Read MCS Records
- Simulate each instruction
- Print changed registers
- See Assembler manual for uploading and downloading instructions

- 2 group members develop thin-slice assembler and simulator
- Hand-over to remaining members for bringing it upto the –bl design
- This way, everyone gets to learn abt assemblers and simulators

What you have to demonstrate and Upcoming Lectures

- Tuesday 9/12 – will demo various SDRAM, Dualport, and VGA Circuits
  - The “find max” program and “try sdram out” details
    - Place results of your experiments on the web
    - For SDRAM
      - Understand and modify overall design; then type up documentation
        - Clocking of Mock CPU
        - Single Pulser
        - Metastability Filter
        - How random testing works
        - How the memtest.vhd waits for more commands after random testing
        - How you modified mock_cpu to do a different set of tests
- A Block-RAM assignment – to be posted on the web

Overall Organization of CPU



Tue 9/12's class + other stuff

- Will go thru the two ckts online already
  - sdramtst50_pipe_memtest_fixed.zip
    - You should have played with this by 9/12
      - Also read application note from www.xess.com
      - Details to be reflected in your documentation
      - We will go over some details
  - dualport_vga_test.zip
    - You should read two more application notes at xess.com
      - VGA generator
      - Dualport unit
    - We will propose experiments for you to report on 9/19 in 1-1 meetings
- Now for the register file circuit
  - You will be given a working Fibonacci circuit
  - You are asked to modify it to a find-max circuit

Show regfile.pdf, fib1.pdf, and fib2.pdf

- You will get to know timings for accessing LD16, ADSU16, and RAMB_S16_S16
- These “parts” are available in the schematic mode of ISE
- Can initialize RAMB4_S16_S16 to contain 0 in location 0 and 1 in location 1
- Fibonacci computation starts wrt it (fib2)
- Fib1 simply swaps locations 0 and 1
- Regfile simply shows how to access RAMB4_S16_S16

Circuit you have to design

- Load RAMB4_S16_S16 with these values at addresses 0, 1, 2…
  - 1,8,2,7,3,6,4,5
- Device a circuit that sweeps an address register AddrReg thru addresses 0 thru 7 remembering the max value seen so far
- After the sweep, AddrReg must be left pointing to the location containing this max
- This circuit is due 9/19 during our 1-1s