Where we are now…

• What we are covering today:
  – RTL design examples, RTL critical path analysis, CPU design

• CAPEs are out!!! https://cape.ucsd.edu/students/
  – Your feedback is very important, please take the time to fill out the survey. I read all your feedback carefully and use it to guide the design of future courses. 😊
  – If at least 250 students do CAPES, I will drop the lowest quiz grade!

• Deadlines:
  – HW#6 due today – the last HW!!!
  – Exam#3 during finals week – the last exam!!!
    • 80 minutes long, Comprehensive
    • Bring one 8 ½ x 11” paper with handwritten notes, but nothing else
    • Sample midterm 3 has been posted:
      – Problem 4: We did not cover the three guidelines heuristic, all else is ok
      – Problem 5: PLA we did not cover, so just skip it
  – Extra prof. office hour during finals week on Monday 1:30-2:30pm
  – Extra TA/tutor office hours starting this week on Friday morning through Tuesday at 11:30am
Data vs. Control Dominated RTL Design

- Data dominant design: extensive datapath, simple controller
- Control dominant design: complex controller, simple datapath

Example of data dominant design: simple filter
Converts digital input stream to new digital output stream
- e.g.: remove noise
  - 180, 180, 181, 180, 240, 180, 181
  - 240 is probably noise, filter might replace by 181
- Simple filter: output average of the last $N$ values
  - Small $N$: less filtering
  - Large $N$: more filtering, but less sharp output

```
X 12
clk
```

```
Y 12
digital filter
```
Data Dominated RTL Design Example: FIR Filter

- **FIR filter**
  - “Finite Impulse Response”
  - A configurable weighted sum of past input values
  - \( y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2) \)
    - Above known as “3 tap”
    - Tens of taps more common
    - Very general filter – User sets the constants \( c_0, c_1, c_2 \) to define a specific filter

- **RTL design**
  - Step 1: Create HLSM
    - Very simple states/transitions

**RTL Design Example**

\[
y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2)
\]

**Assumes constants set to 3, 2, and 2**
FIR Filter: Create datapath

- Begin by creating chain of xt registers to hold past values of X
- Instantiate registers for c0, c1, c2
- Instantiate multipliers to compute c*x values
- Instantiate adders
- Add circuitry to allow loading of particular c register

\[ y(t) = c0 \cdot x(t) + c1 \cdot x(t-1) + c2 \cdot x(t-2) \]

Step 3 & 4: Connect to controller, Create FSM: No controller needed
FIR Filter: Design the Circuit

- Create datapath
- Connect Ctrlr/DP
- Derive FSM
  - Set clr and ld lines appropriately

Datapath for 3-tap FIR filter

Inputs: $X$ (12 bits), Outputs: $Y$ (12 bits)
Local storage: $x_0, x_1, x_2, c_0, c_1, c_2$ (12 bits); $Y_{reg}$ (12 bits)
Comparing the FIR circuit to a software implementation

- **Circuit**
  - Adder has 2-gate delay, multiplier has 20-gate delay
  - Longest path goes through one multiplier and two adders
    - \(20 + 2 + 2 = 24\)-gate delay
  - 100-tap filter, would have about a 34-gate delay: 1 multiplier and 7 adders on longest path

- **Software**
  - 100-tap filter: 100 multiplications, 100 additions.
  - If 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
  - \((100*2 + 100*2)*10 = 4000\) gate delays
RTL: Determining Clock Frequency

- Frequency limited by *longest register-to-register delay*
  - Known as the *critical path*
  - There are more components to the critical path: wire delays, setup/hold constraints, etc.

- Longest path is 7 ns
- Fastest frequency
  - 1 / 7 ns = 142 MHz
RTL: A Circuit May Have Numerous Paths

- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths

- Timing analysis tools need to evaluate all possible paths
RTL Summary

- Datapath and Control Design
- RTL Design Steps
  1. Define the high level state machine
  2. Create datapath
  3. Connect datapath with control
  4. Implement the FSM
- Timing analysis – critical path in more complex circuits
  - Watch out for all possible long paths (e.g. datapath to FSM, FSM control logic, datapath logic etc)
CSE140: Components and Design Techniques for Digital Systems

Single Cycle CPU Design

Tajana Simunic Rosing
CPU Components

• Combinational logic:
  – Boolean equations, logic gates
  – Multiplexors and decoders
  – ALU: executes arithmetic /logical operations
2-input, 32-bit MUX

- Selects one input as the output
Decoder

- 2 input, $2^2 = 4$ outputs

<table>
<thead>
<tr>
<th>I1</th>
<th>I0</th>
<th>O3</th>
<th>O2</th>
<th>O1</th>
<th>O0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Implementation:

Translates input into binary number B and turns on output B
Full 32-bit ALU

**OP CODE**

Performs:
AND, OR, NOT,
ADD, SUB,
Overflow Detection, GTE
MSB ALU

- **Binvert**
- **A31**
- **B31**

- ** CarryIn**
- **OP**

- **ADD**

- **result**

- **GTEin = 0**

- **If GTEout = 1, A ≥ B**

- **CarryOut**

- **GTEout**

- **xor**

- **overflow**

**Equation:**

\[ \text{OP} = \begin{cases} 
0 & \text{if } \text{result} = 0 \\
1 & \text{if } \text{result} = 1 \\
2 & \text{if } \text{result} = 2 \\
3 & \text{if } \text{result} = 3 \\
4 & \text{if } \text{result} = 4 
\end{cases} \]
CPU Components

• Combinational logic:
  – Boolean equations, logic gates
  – Multiplexors and decoders
  – ALU: executes arithmetic /logical operations

• Sequential logic:
  – Storage (memory) elements
  – Counters
Memory elements: D-Latch

- Sets SR-latch (Q) to value of D when clock (C) is high; otherwise last Q retained

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
<th>Reset</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

stores 0
stores 1

R: reset
S: set

Stored state value

Q
¬Q
Memory elements: Flip-Flop

- Stores new value of D in Q when C falls, otherwise current stored value of Q is retained: *falling edge-triggered FF*

![Flip-Flop Diagram]

- C (clock)
- D (data)
- Q (output)
Read/Write Register File

- Input Read Reg #. MUX selects Q for that set of FFs as output.
- Input Write Reg # and Value. Write Value goes to each FF. Write Reg # turns on C to only 1 FF, where Value is stored.

[Diagram of Read/Write Register File]

Sources: TSR, Cummings, KFR
Comparing Processor Memory

- **Register file**
  - Intermediate data storage within CPU
  - Fastest
  - Biggest area/cell
- **SRAM**
  - Fast
  - More compact
  - Used for caches
- **DRAM**
  - Slowest but very compact
    - And refreshing takes time
  - Different technology due to large caps.
  - Used for main memory

```
MxN Memory implemented as a:
```

Size comparison for the same number of bits (not to scale)
• Similar internal structure as register file
  – Decoder enables appropriate word based on address inputs
  – rw controls whether cell is written or read
Static RAM (SRAM) - writing

“Static” RAM cell
- 6 transistors (recall inverter is 2 transistors)
- Writing this cell
  - *word enable* input comes from decoder
  - When 0, value \( d \) loops around inverters
    - That loop is where a bit stays stored
  - When 1, the *data* bit value enters the loop
    - *data* is the bit to be stored in this cell
    - *data’* enters on other side
    - Example shows a “1” being written into cell
Static RAM (SRAM) - reading

- “Static” RAM cell - reading
  - When rw set to read, the RAM logic sets both data and data’ to 1
  - The stored bit d will pull either the left line or the right bit down slightly below 1
  - “Sense amplifiers” detect which side is slightly pulled down
Dynamic RAM (DRAM)

- "Dynamic" RAM cell
  - 1 transistor (rather than 6)
  - Relies on large capacitor to store bit
    - Write: transistor conducts, data voltage level gets stored on top plate of capacitor
    - Read: look at the value of \( d \)
    - Problem: Capacitor discharges over time
      - Must "refresh" regularly, by reading \( d \) and then writing it right back
Memory Storage Permanence

- Traditional ROM/RAM
  - ROM
    - read only, bits stored without power
  - RAM
    - read and write, lose stored bits without power
- Distinctions blurred
  - Advanced ROMs can be written to
    - e.g., EEPROM, FLASH
  - Advanced RAMs can hold bits without power
    - e.g., NVRAM

Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).
ROM & Non-volatile memory

- **Erasable Programmable ROM (EPROM)**
  - Uses “floating-gate transistor” in each cell
  - Programmer uses higher-than-normal voltage so electrons *tunnel* into the gate
    - Electrons become trapped in the gate
    - Only done for cells that should store 0, rest are 1
  - To erase, shine ultraviolet light onto chip

- **Electronically-Erasable Programmable ROM (EEPROM)**
  - Programming similar to EPROM
  - Erasing one word at a time *electronically*

- **Flash memory**
  - Large blocks can be erased *simultaneously*

- **Non-volatile memory (NVM):**
  - **Phase-change memory (PCM)**
    - Material changes phase (liquid to solid) to program
  - **STT-RAM & MRAM**
    - Uses magnetic properties to program
    - Similar to RAM, but with slower writes
CPU Components

• Combinational logic:
  – Boolean equations, logic gates
  – Multiplexors and decoders
  – ALU: executes arithmetic /logical operations

• Sequential logic:
  – The clock
  – Storage (memory) elements
  – Counters

• Datapath and Control: logic block that executes machine language instructions
Control and Datapath Execute Instruction Set

Control takes program as input; it interprets each instruction and tells the Datapath to operate on data via ALU, memory and registers.
**CPU Components – Single Cycle Execution**

Assumptions:
- Every machine language instruction happens in 1 Clock Cycle
- MIPS architecture
  - *Microprocessor without interlocked pipeline stages*
  - reg-reg architecture: all operands must be in registers (total 24)
  - 3 Instruction Types; each instruction 32 bits long
    1. R-type: all data in registers (most arithmetic and logical)
       - e.g. add $s1, $s2, $s3
    2. J-type: jumps and calls
       - e.g. j Label;
    3. I-type: branches, memory transfers, constants
       - e.g. beq $s1, $s2, Label; lw $s1, 32($s2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0, $s1, $s2</td>
<td>000000 10001 10010 10000 00000 100000</td>
</tr>
<tr>
<td></td>
<td>0  17  18  16  0  32</td>
</tr>
</tbody>
</table>
When an assembly language program is run:

- Is assembled, linked, loaded into instruction memory
- PC initialized to the address of the first instruction
  - the PC is really a 32-bit register
  - “counting” is done by separate adders
- rest of clock cycle used to fetch/execute I, update PC
R-type Instruction: reg-reg ALU ops (e.g. add, and)

Tells operation to be performed

Tells specific variant of operation (e.g. add/sub have same opcode)

**R-type Instruction**

<table>
<thead>
<tr>
<th>OPCODE = 0</th>
<th>RS</th>
<th>RT</th>
<th>RD</th>
<th>shamt</th>
<th>FUNCT = 32 or 34</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-11</td>
<td>10-6</td>
<td>5-0</td>
</tr>
</tbody>
</table>

**ADD $S1, $S2, $S3**

**ADD RD, RS, RT**

Source Register 1 (attached to “Read Register 1” input)

Source Register 2 (attached to “Read Register 2” input)

Destination Register (attached to “Write Register” input)

Shift amount (for sll, srl etc.)
Step 1 (R-type): Fetch instruction and advance PC
Step 2 (R-type): Read two registers and set control signals
Step 3 (R-type): Perform the ALU operation
Step 4 (R-type): Write result to register.
### I-Type: Store Instruction

**Tells operation to be performed**

<table>
<thead>
<tr>
<th>Store Instruction</th>
<th>OPCODE = 35 or 43</th>
<th>RS</th>
<th>RT</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

**Base Address Register**
(attached to “Read Register 1” input)

**Source register**
whose value will be stored to memory
(attached to “Read Register 2” input)

**Constant offset**
(added to the base address in RS)

**Example Instructions**

- SW $S1, 32($S2)
- SW RT, #(RS)

**Note:** same as x86

- `MOV [ebx+32], eax`
Step 1 (store): Fetch instruction and advance PC
Step 2 (store): Read register values and set control signals
Step 3 (store): Compute the address
Step 4 (store): Write the value to memory
I-Type: Conditional Branch

<table>
<thead>
<tr>
<th>OPCODE = 4 or 5</th>
<th>RS</th>
<th>RT</th>
<th>BRANCH TARGET’S OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

**BEQ/BNE Instruction**

Source Register 1 (attached to “Read Register 1” input)

Source register 2 (attached to “Read Register 2” input)

Word Offset, which we multiply by 4 (via $\ll 2$) to get Bit Offset, then add to PC+4 to get the address of the instruction to which we branch if RS = RT)

“PC-relative address”

**BEQ** Source1, Source2, Offset

**BEQ** $S1$, $S2$, 100 = AL

4 17 18 25 = ML (in binary)
Step 1 (beq): Fetch instruction and advance PC
Step 2 (beq): Read register values and set control signals.
Step 3 (beq): Compare registers, calculate branch target, and choose new PC
### J-Type: Unconditional Branch

<table>
<thead>
<tr>
<th>JMP/JAL Instruction</th>
<th>OPCODE = 2 or 3</th>
<th>BRANCH TARGET ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31-26</td>
<td>25-0</td>
</tr>
</tbody>
</table>

- **J Offset**
- **J 10000 = AL**
- **2 2500 = ML (in binary)**

Actual Address (in words) which we multiply by 4 (<<2) to get 28-Bit Address, then concatenate to upper 4 bits of PC+4 to get the 32-bit address of the instruction to which we branch unconditionally.
Single-Cycle Datapath with Support for the Jump Instruction