The MIPS Datapath
The Critical Path and Circuit Timing

• The critical path is the slowest path through the circuit

• For a synchronous circuit, the clock cycle **must be longer** than the critical path otherwise we will have timing problems

• It's the combination of the CLK->Q time for the flip flop, the combinational delay, and the setup time
  
  • CLK->Q: Time it takes from when the clock edge comes to the flip-flop output changes
  
  • Combinational delay: The slowest/longest path in the combinational logic
  
  • Setup time: Time that the flip-flop requires the the input be stable before the clock comes
And Hold Time...

• Flip-flops are delicate little circuits...
  • They don't like their input changing before the clock comes (setup time)
  • And they don't like their input changing just after the clock edge (hold time)

• Hold time violations occur when there exists a path through the circuit that is very short
  • CLK->Q + shortest combinational delay < hold time
    • Fix by deliberately adding in delays
How A D-type Flip-Flop Works

• A Flip-Flop is two transparent D-type latches
  • A latch passes D->Q when C is high
  • A latch holds Q when C is low

Latch Truth Table

<table>
<thead>
<tr>
<th>D</th>
<th>CLK</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>NC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This is why we need:
Setup time
Hold Time
CLQ->Q time
Arithmetic and Logic Unit

- Most processors contain a special logic block called the “Arithmetic and Logic Unit” (ALU)
- We’ll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR

![ALU Diagram]

when $S=00$, $R=A+B$
when $S=01$, $R=A-B$
when $S=10$, $R=A \text{ AND } B$
when $S=11$, $R=A \text{ OR } B$
Our simple ALU
Extremely Clever Subtractor

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

XOR serves as conditional inverter!
Clickers – How was the midterm?

Emotion Spectrum

CS61C MIDTERM 1
Clicker Question

Convert the truth table to a boolean expression (no need to simplify):

A: \( F = xy + x(\neg y) \)

B: \( F = xy + (\neg x)y + (\neg x)(\neg y) \)

C: \( F = (\neg x)y + x(\neg y) \)

D: \( F = xy + (\neg x)y \)

E: \( F = (x+y)(\neg x+\neg y) \)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>F(x,y)</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Components of a Computer

- Processor
  - Control
  - Datapath
    - Registers
    - Arithmetic & Logic Unit (ALU)
- Memory
  - Program
  - Bytes
  - Data
- Input
- Output
- Processor-Memory Interface
- I/O-Memory Interfaces
The CPU

- Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making)
- Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)
- Control: portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)
Datapath and Control

- Datapath designed to support data transfers required by instructions
- Controller causes correct transfers to happen
Five Stages of Instruction Execution

- Stage 1: Instruction Fetch
- Stage 2: Instruction Decode
- Stage 3: ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access
- Stage 5: Register Write

- Later on we will add pipeline registers:
  So when one instruction is in stage 3, the next instruction is in stage 2 and the one after that in stage 1
Stages of Execution on Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
There is a wide variety of MIPS instructions: so what general steps do they have in common?

Stage 1: Instruction Fetch

- No matter what the instruction, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy).
- Also, this is where we increment PC (that is, PC = PC + 4, to point to the next instruction: byte addressing so + 4).
- Which is why the branches are all relative to PC + 4, not PC.
Stages of Execution (2/5)

• **Stage 2: Instruction Decode**
  - upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  - first, read the opcode to determine instruction type and field lengths
  - second, read in data from all necessary registers
    - for `add`, read two registers
    - for `addi`, read one register
    - for `jal`, no reads necessary
Stages of Execution (3/5)

- Stage 3: ALU (Arithmetic-Logic Unit)
  - the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)

- what about loads and stores?
  - `lw $t0, 40($t1)`
    - the address we are accessing in memory = the value in $t1 PLUS the value 40
    - so we do this addition in this stage
Stages of Execution (4/5)

- **Stage 4: Memory Access**
  - actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
  - since these instructions have a unique step, we need this extra stage to account for them
  - as a result of the cache system, this stage is expected to be fast
  - But if the cache fails to hit (more later in the class), this can "stall"
Stages of Execution (5/5)

- Stage 5: Register Write
- most instructions write the result of some computation into a register
- examples: arithmetic, logical, shifts, loads, slt
- what about stores, branches, jumps?
  - don’t write anything into a register at the end
  - these remain idle during this fifth stage or skip it all together
Stages of Execution on Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
Datapath Walkthroughs (1/3)

- **add $r3,$r1,$r2 # r3 = r1+r2**
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is an add, then read registers $r1$ and $r2$
  - Stage 3: add the two values retrieved in Stage 2
  - Stage 4: idle (nothing to write to memory)
  - Stage 5: write result of Stage 3 into register $r3$
Example: add Instruction

```
add r3, r1, r2
```

Diagram:
- Instruction memory
- Registers
- ALU
- Data memory

**Flow:**
1. PC
2. instruction memory
3. registers
   - reg[1]
   - reg[2]
4. ALU
5. Data memory
Datapath Walkthroughs (2/3)

- `slti $r3,$r1,17`  
  # if (r1 < 17 ) r3 = 1 else r3 = 0  
- Stage 1: fetch this instruction, increment PC  
- Stage 2: decode to determine it is an `slti`, then read register `$r1`  
  - We will also likely fetch `$r3` but value is ignored:  
    Simplifies decoding logic on deciding which registers to fetch  
- Stage 3: compare value retrieved in Stage 2 with the integer 17  
- Stage 4: idle  
- Stage 5: write the result of Stage 3 (1 if reg source was less than signed immediate, 0 otherwise) into register `$r3`
Example: slti Instruction
Datapath Walkthroughs (3/3)

- `sw $r3,16($r1) # Mem[r1+16]=r3`

  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine it is a `sw`, then read registers `$r1` and `$r3`
  - Stage 3: add 16 to value in register `$r1` (retrieved in Stage 2) to compute address
  - Stage 4: write value in register `$r3` (retrieved in Stage 2) into memory address computed in Stage 3
  - Stage 5: idle (nothing to write into a register)
Example: sw Instruction

\[
\text{MEM}[r1+17] = r3
\]
Why Five Stages? (1/2)

- Could we have a different number of stages?
- Yes, other ISAs have different natural number of stages
  - And these days, pipelining can be much more aggressive than the "natural" 5 stages MIPS uses
- Why does MIPS have five if instructions tend to idle for at least one stage?
  - Five stages are the union of all the operations needed by all the instructions.
  - One instruction uses all five stages: the load
Why Five Stages? (2/2)

• `lw $r3,16($r1) # r3=Mem[r1+16]`
  • Stage 1: fetch this instruction, increment PC
  • Stage 2: decode to determine it is a `lw`, then read register `$r1`
  • Stage 3: add 16 to value in register `$r1` (retrieved in Stage 2)
  • Stage 4: read value from memory address computed in Stage 3
  • Stage 5: write value read in Stage 4 into register `$r3`
Example: lw Instruction

```
lw r3, 17(r1)
```

PC -> instruction memory -> registers -> ALU -> Data memory

- lw r3, 17(r1) -> reg[1] + 16
- MEM[r1+16]
Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements
   • Meaning of each instruction is given by register transfers
   • Datapath must include storage element for ISA registers
   • Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
The MIPS Instruction Formats

All MIPS instructions are 32 bits long. 3 formats:

- **R-type**
  - op: operation ("opcode") of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field

- **I-type**
  - op: operation ("opcode") of the instruction
  - rs, rt: the source and destination register specifiers
  - address/immediate: address offset or immediate value

- **J-type**
  - op: operation ("opcode") of the instruction
  - target address of jump instruction

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address/immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
The MIPS-lite Subset

- **ADDU and SUBU**
  - `addu rd,rs,rt`
  - `subu rd,rs,rt`
- **OR Immediate:**
  - `ori rt,rs,imm16`
- **LOAD and STORE Word**
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`
- **BRANCH:**
  - `beq rs,rt,imm16`
Register Transfer Level (RTL)

- Colloquially called “Register Transfer Language”
- RTL gives the meaning of the instructions
- All start by fetching the instruction itself

\[
\begin{align*}
&\{op, rs, rt, rd, sham, funct\} \leftarrow MEM[PC] \\
&\{op, rs, rt, \text{Imm16}\} \leftarrow MEM[PC] \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>( R[rd] \leftarrow R[rs] + R[rt]; PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>SUBU</td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>ORI</td>
<td>( R[rt] \leftarrow R[rs]</td>
</tr>
<tr>
<td>LOAD</td>
<td>( R[rt] \leftarrow MEM[ R[rs] + \text{sign_ext(Imm16)}]; PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[ R[rs] + \text{sign_ext(Imm16)}] \leftarrow R[rt]; PC \leftarrow PC + 4 )</td>
</tr>
</tbody>
</table>
| BEQ  | if ( \( R[rs] == R[rt] \) )  \\
|      | \( PC \leftarrow PC + 4 + \{\text{sign\_ext(Imm16)}, 2'b00\} \)  \\
|      | else \( PC \leftarrow PC + 4 \) |
In Conclusion

- “Divide and Conquer” to build complex logic blocks from smaller simpler pieces (adder)
- Five stages of MIPS instruction execution
- Mapping instructions to datapath components
- Single long clock cycle per instruction
If There is Time Left: CloudBleed...

- How many got "change your password" emails from various services?
- What happened: C out-of-bounds read!
  - `if(ptr == end)...`
    - Unfortunately, `ptr` got incremented 1 past end when it couldn't parse HTML right
    - Result was that Cloudflare would, under some conditions, read out memory
- This memory contained other requests to different cloudflare sites
  - And it would get picked up by search engines and stored in their caches!
Lessons Learned...

• Without detailed audibility, it's impossible to know what was lost
  • Which is why some sites went "oh, change your password", even though the odds of your password being compromised are slim
Lessons Already Forgotten...

• Don't Use C/C++!
  • This is "yet another massively disruptive bug caused because programmers don't use safe languages!"

• And if you do, don't use pointer arithmetic
  • Array logic, eg if (index >= end) ... rather less likely to screw up

• Use Valgrind or similar
  • Valgrind would have trapped this as an "out of bounds read"