CS 61C:  
Great Ideas in Computer Architecture  
*MIPS Datapath*  

Instructors:  
Nicholas Weaver & Vladimir Stojanovic  
http://inst.eecs.Berkeley.edu/~cs61c/sp16
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.

- 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
How to design Adder/Subtractor?

- Truth-table, then determine canonical form, then minimize and implement as we’ve seen before
- Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer
Adder/Subtractor – One-bit adder
LSB... (Half Adder)

\[ \begin{array}{ccc}
    a_3 & a_2 & a_1 \\
    b_3 & b_2 & b_1 \\
    s_3 & s_2 & s_1 \\
\end{array} \]

\[ \begin{array}{c|c|c|c}
    a_0 & b_0 & s_0 & c_1 \\
    \hline
    0 & 0 & 0 & 0 \\
    0 & 1 & 1 & 0 \\
    1 & 0 & 1 & 0 \\
    1 & 1 & 0 & 1 \\
\end{array} \]

\[ s_0 = \]
\[ c_1 = \]
Adder/Subtractor – One-bit full-adder (1/2)…

<table>
<thead>
<tr>
<th></th>
<th>(a_i)</th>
<th>(b_i)</th>
<th>(c_i)</th>
<th>(s_i)</th>
<th>(c_{i+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  s_i &= \\
  c_{i+1} &= 
\end{align*}
\]
Adder/Subtractor – One-bit adder (2/2)

\[ s_i = \text{XOR}(a_i, b_i, c_i) \]
\[ c_{i+1} = \text{MAJ}(a_i, b_i, c_i) = a_ib_i + a_ic_i + b_ic_i \]
N 1-bit adders $\Rightarrow$ 1 N-bit adder

What about overflow? 
Overflow $= c_n$?
Extremely Clever Adder/Subtractor

XOR serves as conditional inverter!
Aka "Subtract is Invert and add 1"

<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>
Clicker Question

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

A: $F = xy + x(y)$

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

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

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

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

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

• Project 2-2 is out!
Components of a Computer

- Processor
- Control
- Datapath
  - PC
  - Registers
  - Arithmetic & Logic Unit (ALU)
- Memory
  - Program
  - Bytes
  - Data
- Processor-Memory Interface
- I/O-Memory Interfaces
- Input
- Output
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
Stages of Execution on Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
Stages of Execution (1/5)

• 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)
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
• 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
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

Instruction: add r3, r1, r2

PC +4 -> instruction memory

 register

imm

ALU:

Data memory

+4
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
  – 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

slti $r3, $r1, 17

$reg[1] < 17?
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

```
sw r3, 16(r1)
```

```
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)
```

```
MEM[r1+16]
```
Clickers/Peer Instruction

• Which type of MIPS instruction is active in the fewest stages?

A: LW
B: BEQ
C: J
D: JAL
E: ADDU
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:

<table>
<thead>
<tr>
<th>Field</th>
<th>R-type</th>
<th>L-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit position</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>bit length</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>bit length</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>bit length</td>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>bit length</td>
<td>6</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

- **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`, `rd`: the source and destination register specifiers
  - `address/immediate`: address offset or immediate value

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

The different fields are:
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

\[
\begin{aligned}
\{ op, rs, rt, rd, shamt, funct \} & \leftarrow MEM[ PC ] \\
\{ op, rs, rt, \text{ Imm16} \} & \leftarrow MEM[ PC ] \\
\end{aligned}
\]

**Inst**  **Register Transfers**

**ADDU**  \( R[rd] \leftarrow R[rs] + R[rt]; \)  PC  \( \leftarrow PC + 4 \)

**SUBU**  \( R[rd] \leftarrow R[rs] - R[rt]; \)  PC  \( \leftarrow PC + 4 \)

**ORI**  \( R[rt] \leftarrow R[rs] | \)  zero_ext(Imm16);  PC  \( \leftarrow PC + 4 \)

**LOAD**  \( R[rt] \leftarrow MEM[ R[rs] + \text{ sign_ext(Imm16)} ]; \)  PC  \( \leftarrow PC + 4 \)

**STORE**  \( \text{MEM[ } R[rs] + \text{ sign_ext(Imm16)} ] \leftarrow R[rt]; \)  PC  \( \leftarrow PC + 4 \)

**BEQ**  if ( \( R[rs] == R[rt] \) )

\[
\begin{aligned}
\text{PC} & \leftarrow PC + 4 + \{ \text{sign_ext(Imm16)}, 2\text{'b00} \} \\
\text{else} \text{ PC} & \leftarrow PC + 4
\end{aligned}
\]
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