Levels of Representation/Interpretation

- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., MIPS)
- Machine Language Program (MIPS)

Compiler

Assembler

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

```plaintext
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $t0, 0($2)
lw  $t1, 4($2)
sw  $t1, 0($2)
sw  $t0, 4($2)
```

Anything can be represented as a number, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Review

• D-Flip-Flops update output at rising edge of clock
  – Setup and Hold times important
• Critical Path constrains clock rate.
• Finite State Machines extremely useful
• Use muxes to select among input
  – S input bits selects $2^S$ inputs
  – Each input can be n-bits wide, independent of S
• Build n-bit adder out of chained 1-bit adders.
Review: N x 1-bit Adders ⇒ 1 N-bit Adder

Connect Carry Out i-1 to Carry in i:
What about detecting overflow?

- Unsigned overflow - The carry out from the most significant bit.

- Signed overflow - A bit more complicated, two cases:
  - Overflow from adding “big” positive numbers.
  - Overflow from adding “big” negative numbers.
Detecting Signed Overflow (4-bit examples)

• From two positive numbers:
  • 0111 + 0111, 0111 + 0001, 0100 + 0100.
  • What do these have in common? Carry-out from the second highest bit (but not highest bit)

• From two negative numbers:
  • 1000 + 1000, 1000 + 1111, 1011 + 1011.
  • These have carry-out from the highest bit (but not second highest bit)

• Expression for signed overflow: $C_n \text{ XOR } C_{n-1}$
Twos Complement Adder/Subtractor

Can subtract by adding the negative of the second number. To negate:

Flip the bits

And add one!
Agenda

• 5 Stages of the Datapath
• Administrivia
• Quick Datapath Walkthrough
• Processor Design Process
  – Determine datapath requirements based on instruction set
  – Select datapath components
  – Assemble the datapath
Five Components of a Computer

- Control
- Datapath
- Memory
- Input
- Output
The Processor

- **Processor (CPU)**: Implements the instructions of the Instruction Set Architecture (ISA)
- **Datapath**: portion of the processor which contains hardware necessary to perform operations required by the processor (the brawn)
- **Control**: portion of the processor (also in hardware) which tells the datapath what needs to be done (the brain)
Stages of the Datapath: Overview

• Break up the process of “executing an instruction” into stages or phases, and then connect the phases to create the whole datapath
  – Smaller phases are easier to design
  – Easy to optimize (change) one phase without touching the others

• Project 1 had 3 phases: Fetch, Decode, Execute. Here, we expand Execute into ALU, Memory Access, and Register Write.
Phases of the Datapath (1/5)

• Phase 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)
• Phase 2: *Instruction Decode*
  
  – Upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  – Read the opcode and each of the possible fields from the 32 bits of the instruction.
  – Read data from all necessary registers (This differs from Project 1)
    • For add, read two registers
    • For addi, read one register
    • For jal, no reads necessary
Phases of the Datapath (3/5)

• Phase 3: *ALU* (Arithmetic-Logic Unit)
  – Real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)
  – What about loads and stores?
    • eg, lw $t0, 40($t1)
    • Memory Address = Offset + Value in $t1
    • We perform this addition in this stage.
Phases of the Datapath (4/5)

• Phase 4: *Memory Access*
  – Only the load and store instructions do anything during this phase; the others remain idle or skip this phase all together
  – Since these instructions have a unique step, we need this extra phase to account for them
  – As a result of the cache system, this phase is expected to be fast
Phases of the Datapath (5/5)

• Phase 5: Register Write
  – Most instructions write the result of some computation into a register
  – E.g.,: arithmetic, logical, shifts, loads, slt, jal(!)
  – What about stores, branches, jumps?
    • Don’t write anything into a register at the end
    • These remain idle during this fifth phase or skip it all together
Why Five Stages?

• Could we have a different number of stages?
  – Yes, and other architectures do

• So why does MIPS have five if instructions tend to idle for at least one stage?
  – The five stages are the union of all the operations needed by all the instructions.
  – There is one instruction that uses all five stages: the load
Generic Steps of Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Reg. Write
Agenda

• 5 Stages of the Datapath
• Administrivia
• Quick Datapath Walkthrough
• Processor Design Process
  – Determine datapath requirements based on instruction set
  – Select datapath components
  – Assemble the datapath
**Administrivia**

- HW3 Due Wednesday at midnight
  - Should be able to answer last two questions after today’s lecture.
- Project 2 Part 2 due Sunday.
- Lab 11 posted.
- Lab 12 cancelled!
  - Replaced with free study session where you can catch up on labs / work on project 2.
  - The TA’s will still be there.
cs61c in...Minecraft!

• Person builds 16-bit ALU in Minecraft.

• Let’s check out the video...
Agenda

• 5 Stages of the Datapath
• Administrivia
• Quick Datapath Walkthrough
• Processor Design Process
  – Determine datapath requirements based on instruction set
  – Select datapath components
  – Assemble the datapath
Datapath Walkthroughs (1/3)

  - Stage 1: fetch this instruction, inc. PC  
  - Stage 2: decode to find it’s an `add`, then read registers $1 and $2  
  - 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 $3
Example: \texttt{add} Instruction

\texttt{add $3, $1, $2}
Datapath Walkthroughs (2/3)

- `slti   $3,$1,17`
  - Stage 1: fetch this instruction, inc. PC
  - Stage 2: decode to find it’s an `slti`, then read register $1
  - Stage 3: compare value retrieved in Stage 2 with the integer 17
  - Stage 4: idle
  - Stage 5: write the result of Stage 3 in register $3
Example: `slti` Instruction

```assembly
slti $3, $1, 17
```

Diagram:
- Instruction memory
- Registers
- ALU
- Data memory
- PC
- Immediate value `+4`
Datapath Walkthroughs (3/3)

- **sw $3, 17($1)**
  - Stage 1: fetch this instruction, inc. PC
  - Stage 2: decode to find it’s a sw, then read registers $1 and $3
  - Stage 3: add 17 to value in register $1 (retrieved in Stage 2)
  - Stage 4: write value in register $3 (retrieved in Stage 2) into memory address computed in Stage 3
  - Stage 5: idle (nothing to write into a register)
Example: $w$ Instruction

\[ \text{SW } \$3, 17(\$1) \]

\[ \text{MEM}[$1+17]=\$3 \]
Agenda

• 5 Stages of the Datapath
• Administrivia
• Quick Datapath Walkthrough

• Processor Design Process
  – Determine datapath requirements based on instruction set
  – Select datapath components
  – Assemble the datapath
Processor Design Process

• Five steps to design a processor:
  Step 1: Analyze instruction set to determine datapath requirements
  Step 2: Select set of datapath components & establish clocking methodology
  Step 3: Assemble datapath components to 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
Processor Design Process

- Five steps to design a processor:
  - **Step 1:** Analyze instruction set to determine datapath requirements
  - **Step 2:** Select set of datapath components & establish clocking methodology
  - **Step 3:** Assemble datapath components to 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
Step 1a: The MIPS-lite Subset for today

- **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

---

7/25/2011  Spring 2011 -- Lecture #18
Register Transfer Language (RTL)

- RTL gives the **meaning** of the instructions

\[
\{ \text{op, rs, rt, rd, shamt, funct} \} \leftarrow \text{MEM[ PC ]} \quad \text{Instruction Fetches}
\]

\[
\{ \text{op, rs, rt, Imm16} \} \leftarrow \text{MEM[ PC ]}
\]

<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]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>SUBU</td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>ORI</td>
<td>( R[rt] \leftarrow R[rs] \mid \text{zero_ext}(\text{Imm16}); \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( R[rt] \leftarrow \text{MEM[ R[rs] + sign_ext(Imm16)]]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM[ R[rs] + sign_ext(Imm16)]} \leftarrow R[rt]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
</tbody>
</table>
| BEQ  | if ( \( R[rs] == R[rt] \) )  
|      | then \( \text{PC} \leftarrow \text{PC} + 4 + (\text{sign_ext}(\text{Imm16}) || 00) \)  
|      | else \( \text{PC} \leftarrow \text{PC} + 4 \) |
Step 1b: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (will use one for each: really caches)
- Registers (R: 32 x 32)
  - Read rs
  - Read rt
  - Write rt or rd
- PC
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
- Add 4 (+ maybe extended immediate) to PC
- Compare if registers equal?
Step 2: Components of the Datapath

- Combinational Elements

- Storage Elements (Registers, Memory)
Required ALU Operations

• Addition, subtraction, logical OR, ==:
  ADDU  R[rd] = R[rs] + R[rt]; ...
  SUBU  R[rd] = R[rs] - R[rt]; ...
  ORI   R[rt] = R[rs] | zero_ext(Imm16)...
  BEQ   if ( R[rs] == R[rt] )...

• Test to see if output == 0 for any ALU operation gives == test. How?

• Full MIPS also adds AND, Set Less Than (1 if A < B, 0 otherwise)

• ALU from Appendix C, section C.5
Storage Element: Idealized Memory

• Memory (idealized)
  – One input bus: Data In
  – One output bus: Data Out

• Memory word is found by:
  – Address selects the word to put on Data Out
  – Write Enable = 1: address selects the memory word to be written via the Data In bus

• Clock input (CLK)
  – CLK input is a factor ONLY during write operation
  – During read operation, behaves as a combinational logic block:
    • Address valid \(\Rightarrow\) Data Out valid after “access time”
Storage Element: Register (Building Block)

• Similar to D Flip Flop except
  – N-bit input and output
  – Write Enable input

• Write Enable:
  – Negated (or deasserted) (0): Data Out will not change
  – Asserted (1): Data Out will become Data In on rising edge of clock
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after “access time.”
Clocking Methodology (1/2)

• Single Cycle CPU: All stages of an instruction are completed within one long clock cycle.
  – The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Reg. Write

• This is what we’ll talk about today.
• Multiple-cycle CPU: Only one stage of instruction per clock cycle.
  – The clock is made as long as the slowest stage.
  
  – Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).
Step 3: Assemble Datapath Meeting Requirements

Register Transfer Requirements ⇒ Assembly of Datapath

• In common between all instructions:
  – Fetch the Instruction: mem[PC]
  – Update the program counter:
    • Sequential Code: PC ← PC + 4
    • Branch and Jump: PC ← “something else”

Diagram:
- Clock (clk)
- PC
- Instruction Memory
- Instruction Word
- Next Address Logic
- 32-bit instruction word

7/25/2011  Spring 2011 -- Lecture #18
Add & Subtract

- \( R[rd] = R[rs] \) op \( R[rt] \) (addu \( rd,rs,rt \))
  - \( Ra, Rb, \) and \( Rw \) come from instruction’s Rs, Rt, and Rd fields

- ALUctr and RegWr: control logic after decoding the instruction

- ... Already defined the register file & ALU
Logical Operations with Immediate

- \( R[rt] = R[rs] \text{ op } \text{ZeroExt}[\text{imm16}] \)

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
31 & 26 & 21 & 16 & 15 & 0 \\
\text{6 bits} & \text{5 bits} & \text{5 bits} & \text{16 bits} & \\
0000000000000000 & \text{immediate} \\
\text{16 bits} & \text{16 bits} & \\
\end{array}
\]

But we’re writing to \( R_t \) register??
Logical Operations with Immediate

- \( R[rt] = R[rs] \text{ op } \text{ZeroExt}[imm16] \)

![Diagram of ALU with RegFile, ZeroExt, ALUSrc, RegWr, RegDst, ALUctr, busA, busB, and clk connections.]

- Already defined 32-bit MUX; Zero Ext?
Load Operations

- \( R[rt] = \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)

Example: \( \text{lw} \ rt, rs, \text{imm16} \)

What sign extending??
Load Operations

- \( R[rt] = Mem[R[rs] + SignExt[imm16]] \)

Example: \( lw \ rt, rs, imm16 \)
Store Operations


Ex.: sw rt, rs, imm16

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

RegDst \(\rightarrow\) Rd, Rt

RegWr \(\rightarrow\) Rs, Rt, 5

busW \(\rightarrow\) 32

RegFile

Extender

ALU

MemtoReg

Data Memory
Store Operations

- \[ \text{Mem} [ \text{R}[\text{rs}] + \text{SignExt}[\text{imm16}] ] = \text{R}[\text{rt}] \]
  
  Ex.: \( \text{sw} \ \text{rt}, \ \text{rs}, \ \text{imm16} \)

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

Diagram:

- **RegFile**
  - \( \text{Rw} \), \( \text{Ra} \), \( \text{Rb} \)
  - \( \text{clk} \)
  - \( \text{busW} \) (32)
  - \( \text{imm16} \) (16)
  - \( \text{ExtOp} \)

- **Extender**
  - \( \text{ExtOp} \)
  - 16 bits

- **ALU**
  - \( \text{ALUSrc} \)
  - \( \text{WrEn} \)
  - \( \text{Adr} \)
  - \( \text{Data In} \) (32)
  - \( \text{clk} \)

- **MemtoReg**
  - \( \text{MemWr} \)
  - \( \text{MemtoReg} \)
The Branch Instruction

beq rs, rt, imm16

– mem[PC] Fetch the instruction from memory
– Zero = R[rs] - R[rt] Calculate branch condition
– if (Zero) Calculate the next instruction’s address
  • PC = PC + 4 + ( SignExt(imm16) x 4 )
else
  • PC = PC + 4
Datapath for Branch Operations

- `beq rs, rt, imm16`

Datapath generates condition (zero)

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
```

Inst Address

Already have mux, adder, need special sign extender for PC, need equal compare (sub?)
An Abstract View of the Implementation