Review

• CPU design involves Datapath, Control
  – 5 Stages for MIPS Instructions
    1. Instruction Fetch
    2. Instruction Decode & Register Read
    3. ALU (Execute)
    4. Memory
    5. Register Write

• Datapath timing: single long clock cycle or one short clock cycle per stage
Datapath and Control

- Datapath based on data transfers required to perform instructions
- Controller causes the right transfers to happen
CPU Clocking (1/2)

• For each instruction, how do we control the flow of information though the datapath?
• Single Cycle CPU: All stages of an instruction completed within one long clock cycle
  – Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle

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

- Alternative multiple-cycle CPU: only one stage of instruction per clock cycle
  - 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)
Agenda

• Stages of the Datapath
• Datapath Instruction Walkthroughs
• Datapath Design
Five Components of a Computer

- **Processor**
- **Control**
- **Datapath**
- **Computer (passive)**
  - (where programs, data live when running)
- **Memory**
  - (where programs, data live when running)
- **Devices**
  - **Input**
  - **Output**
- **Keyboard, Mouse**
  - **Disk**
    - (where programs, data live when not running)
- **Display, Printer**
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>I-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs, rt, rd</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>shamt</td>
<td>5 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td>6 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>address/immediate</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>target address</td>
<td>26 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The different fields are:
  - **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
  - **address / immediate**: address offset or immediate value
  - **target address**: target address of jump instruction
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 Language (RTL)

RTL gives the **meaning** of the instructions

All start by fetching the instruction

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

<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 \text{MEM}[ R[rs] + \text{sign_ext}(\text{Imm16})]; \ PC \leftarrow \ PC + 4 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[ R[rs] + \text{sign_ext}(\text{Imm16}) ] \leftarrow R[rt]; \ PC \leftarrow \ PC + 4 )</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( ( R[rs] == R[rt] ) ) then ( PC \leftarrow PC + 4 + (\text{sign_ext}(\text{Imm16})</td>
</tr>
</tbody>
</table>
Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (will use one for each)
- 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 registers?
Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements + Clocking Methodology
- Building Blocks

![Adder Diagram]

- Adder diagram with inputs A and B, 32-bit operands, output sum, carry-in, and carry-out.

![Multiplexer Diagram]

- Multiplexer diagram with select input and 32-bit operands A and B, output Y.

![ALU Diagram]

- ALU diagram with inputs A and B, 32-bit operands, output result.

![OP Input]

- OP input to ALU.
ALU Needs for MIPS-lite + Rest of MIPS

• 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?

• P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)

• ALU follows Chapter 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 positive 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 $\Rightarrow$ busA or busB valid after “access time.”
Step 3a: Instruction Fetch Unit

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: 
    $$\text{mem}[\text{PC}]$$
  - Update the program counter:
    - Sequential Code: 
      $$\text{PC} \leftarrow \text{PC} + 4$$
    - Branch and Jump: 
      $$\text{PC} \leftarrow \text{“something else”}$$
Step 3b: Add & Subtract

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

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

– \( \text{ALUctr} \) and \( \text{RegWr} \): control logic after decoding the instruction

• ... Already defined the register file & ALU
Clocking Methodology

- Storage elements clocked by same edge
- Flip-flops (FFs) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q delay
- “Critical path” (longest path through logic) determines length of clock period
### Register-Register Timing: One Complete Cycle

#### Instruction Memory Access Time

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rs, Rt, Rd, Op, Func</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>

#### Delay through Control Logic

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUctr</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>

#### Register File Access Time

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWr</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>

#### ALU Delay

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>busA, B</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>

#### Register Write

<table>
<thead>
<tr>
<th></th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>busW</td>
<td>Old Value</td>
<td>New Value</td>
</tr>
</tbody>
</table>

---

**Diagram:**

- **RegWr:** 5\[Rw\], 5\[Ra\], 5\[Rb\]
- **RegFile:** 32\[Rd\], 32\[Rs\], 32\[Rt\]
- **ALU:** 32, 32
- **RegWrite Occurs Here:**
Putting it All Together: A Single Cycle Datapath

Inst Memory

nPC_sel

Adder

Mux

RegDst

RegWr

ALUctr

MemtoReg

MemWr

Data Memory

ExtOp

ALUSrc

Imm16

Rd

Rt

Rs

Rt

Rd

Imm16

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>

Instruction<31:0>
Processor Design: 3 of 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