CMSC 611: Advanced Computer Architecture

Instruction Set Architecture & Pipelining
All data in computer systems is represented in binary
Instructions are no exception
The program that translates the human-readable code to numeric form is called an Assembler
Hence *machine-language* or *assembly-language*

Example:

Assembly:  \( \text{ADD } \$t0, \$s1, \$s2 \)

Note: by default MIPS \$t0..\$t7 map to reg. 8..15, \$s0..\$s7 map to reg. 16-23

M/C language (hex by field):  \( 0x0 \ 0x11 \ 0x12 \ 0x8 \ 0x020 \)

M/C language (binary):  \( 00000010001100100100000001000000 \)

M/C language (hex):  \( 0x02324020 \)
Encoding an Instruction Set

• Affects the size of the compiled program
• Also complexity of the CPU implementation
• Operation in one field called opcode
• Addressing mode in opcode or separate field
• Must balance:
  – Desire to support as many registers and addressing modes as possible
  – Effect of operand specification on the size of the instruction (and program)
  – Desire to simplify instruction fetching and decoding during execution
• Fixed size instruction encoding simplifies CPU design but limits addressing choices
## Encoding Examples

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>...</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

(a) Variable (e.g., VAX, Intel 80x86)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)
MIPS Instruction Formats

I-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
</table>

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
</table>

Jump and jump and link
Trap and return from exception

<table>
<thead>
<tr>
<th>opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>R-type</td>
</tr>
<tr>
<td>addi</td>
</tr>
<tr>
<td>llo</td>
</tr>
<tr>
<td>lb</td>
</tr>
<tr>
<td>sb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>sll</td>
</tr>
<tr>
<td>jr</td>
</tr>
<tr>
<td>mfhi</td>
</tr>
<tr>
<td>mult</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>slt</td>
</tr>
<tr>
<td>101</td>
</tr>
</tbody>
</table>
GPU Shading ISA

• Data
  – IEEE-like floating point
  – 4-element vectors
    • Most instructions perform operation on all four

• Addressing
  – No addresses
  – ATTRIB, PARAM, TEMP, OUTPUT
  – Limited arrays
  – Element selection (read & write)
    • C.xyw, C.rgb
### GPU Shading ISA

**Instructions:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS r,s</td>
<td>r = abs(s)</td>
<td>MIN r,s1,s2</td>
<td>r = min(s1,s2)</td>
</tr>
<tr>
<td>ADD r,s1,s2</td>
<td>r = s1+s2</td>
<td>MOV r,s1</td>
<td>r = s1</td>
</tr>
<tr>
<td>CMP r,c,s1,s2</td>
<td>r = c&lt;0 ? s1 : s2</td>
<td>MUL r,s1,s2</td>
<td>r = s1*s2</td>
</tr>
<tr>
<td>COS r,s</td>
<td>r = cos(s)</td>
<td>POW r,s1,s2</td>
<td>r ≈ s1^s2</td>
</tr>
<tr>
<td>DP3 r,s1,s2</td>
<td>r = s1.xyz • s2.xyz</td>
<td>RCP r,s1</td>
<td>r = 1/s1</td>
</tr>
<tr>
<td>DP4 r,s1,s2</td>
<td>r = s1 • s2</td>
<td>RSQ r,s1</td>
<td>r = 1/sqrt(s1)</td>
</tr>
<tr>
<td>DPH r,s1,s2</td>
<td>r = s1.xyz1 • s2</td>
<td>SCS r,s1</td>
<td>r = (cos(s),sin(s),?,?)</td>
</tr>
<tr>
<td>DST r,s1,s2</td>
<td>r = (1,s1.y*s2.y,s1.z,s2.w)</td>
<td>SGE r,s1,s2</td>
<td>r = s1≥s2 ? 1 : 0</td>
</tr>
<tr>
<td>EX2 r,s</td>
<td>r ≈ 2^s</td>
<td>SIN r,s</td>
<td>r = sin(s)</td>
</tr>
<tr>
<td>FLR r,s</td>
<td>r = floor(s)</td>
<td>SLT r,s1,s2</td>
<td>r = s1&lt;s2 ? 1 : 0</td>
</tr>
<tr>
<td>FRC r,s</td>
<td>r = s - floor(s)</td>
<td>SUB r,s1,s2</td>
<td>r = s1-s2</td>
</tr>
<tr>
<td>KIL s</td>
<td>if (s&lt;0) discard</td>
<td>SWZ r,s,cx, cy,cz,cw</td>
<td>r = swizzle(s)</td>
</tr>
<tr>
<td>LG2 r,s</td>
<td>r ≈ log_2(s)</td>
<td>TEX r,s,name,nD</td>
<td>r = texture(s)</td>
</tr>
<tr>
<td>LIT r,s</td>
<td>r = lighting computation</td>
<td>TXB r,s,name,nD</td>
<td>r = textureLOD(s)</td>
</tr>
<tr>
<td>LRP r,t,s1,s2</td>
<td>r = t*s1 + (1-t)*s2</td>
<td>TXP r,s,name,nD</td>
<td>r = texture(s/s.w)</td>
</tr>
<tr>
<td>MAD r,s1,s2,s3</td>
<td>r = s1*s2 + s3</td>
<td>XPD r,s1,s2</td>
<td>r = s1×s2</td>
</tr>
<tr>
<td>MAX r,s1,s2</td>
<td>r = max(s1,s2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GPU Shading ISA

• Notable:
  – Many special-purpose instructions
  – No binary encoding, interface is text form
    • No ISA limits on future expansion
    • No ISA limits on registers
    • No ISA limits on immediate values
  – Originally no branching! (exists now)
Washer takes 30 min, Dryer takes 40 min, folding takes 20 min

Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?
• Pipelining means start work as soon as possible
• Pipelined laundry takes 3.5 hours for 4 loads
• Pipelining doesn’t help latency of single task, it helps throughput of entire workload
• Pipeline rate limited by slowest pipeline stage
• Multiple tasks operating simultaneously using different resources
• Potential speedup = Number pipe stages
• Unbalanced lengths of pipe stages reduces speedup
• Time to “fill” pipeline and time to “drain” it reduce speedup
• Stall for Dependencies
MIPS Instruction Set

• RISC characterized by the following features that simplify implementation:
  – All ALU operations apply only on registers
  – Memory is affected only by load and store
  – Instructions follow very few formats and typically are of the same size
Single-cycle Execution
Multi-Cycle Implementation of MIPS

1. Instruction fetch cycle (IF)
   IR ← Mem[PC];   NPC ← PC + 4

2. Instruction decode/register fetch cycle (ID)
   A ← Regs[IR_6..10];   B ← Regs[IR_11..15];   Imm ← ((IR_16)^16 #+# IR_16..31)

3. Execution/effective address cycle (EX)
   Memory ref: ALUOutput ← A + Imm;
   Reg-Reg ALU: ALUOutput ← A func B;
   Reg-Imm ALU: ALUOutput ← A op Imm;
   Branch: ALUOutput ← NPC + Imm;   Cond ← (A op 0)

4. Memory access/branch completion cycle (MEM)
   Memory ref: LMD ← Mem[ALUOutput] or Mem(ALUOutput) ← B;
   Branch: if (cond) PC ← ALUOutput;

5. Write-back cycle (WB)
   Reg-Reg ALU: Regs[IR_16..20] ← ALUOutput;
   Reg-Imm ALU: Regs[IR_11..15] ← ALUOutput;
   Load: Regs[IR_11..15] ← LMD;
Multi-cycle Execution

Figure: Dave Patterson
• The load instruction is the longest
• All instructions follow at most the following five steps:
  – Ifetch: Instruction Fetch
    • Fetch the instruction from the Instruction Memory and update PC
  – Reg/Dec: Registers Fetch and Instruction Decode
  – Exec: Calculate the memory address
  – Mem: Read the data from the Data Memory
  – WB: Write the data back to the register file
Instruction Pipelining

- Start handling next instruction while the current instruction is in progress
- Feasible when different devices at different stages

Program Flow

Time between instructions\(_{\text{pipelined}}\) = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}

Pipelining improves performance by increasing instruction throughput
Example of Instruction Pipelining

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>Reg</th>
<th>ALU</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 200($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time between first & fourth instructions is $3 \times 8 = 24$ ns

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 200($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time between first & fourth instructions is $3 \times 2 = 6$ ns

Ideal and upper bound for speedup is number of stages in the pipeline
• Cycle time long enough for longest instruction
• Shorter instructions waste time
• No overlap
- Cycle time long enough for longest stage
- Shorter stages waste time
- Shorter instructions can take fewer cycles
- No overlap
- Cycle time long enough for longest stage
- Shorter stages waste time
- No additional benefit from shorter instructions
- Overlap instruction execution
Pipeline Performance

• Pipeline increases the instruction throughput
  – not execution time of an individual instruction
• An individual instruction can be slower:
  – Additional pipeline control
  – Imbalance among pipeline stages
• Suppose we execute 100 instructions:
  – Single Cycle Machine
    • 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
  – Multi-cycle Machine
    • 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
  – Ideal 5 stages pipelined machine
    • 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
• Lose performance due to fill and drain
Pipeline Datapath

- Every stage must be completed in one clock cycle to avoid stalls.
- Values must be latched to ensure correct execution of instructions.
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction).
### Pipeline Stage Interface

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any Instruction</th>
</tr>
</thead>
</table>
| **IF** | IF/ID.IR ← MEM[PC];  
        | IF/ID.NPC, PC ← (if (EX/MEM.opcode == branch) & EX/MEM.cond)  
        | \{EX/MEM.ALUOutput \} else \{ PC + 4 \} ;  |
| **ID** | ID/EX.A = Regs[IF/ID.IR 6..10]; ID/EX.B ← Regs[IF/ID.IR 11..15];  
        | ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;  
        | ID/EX.Imm ← (IF/ID.IR 16) 16 # IF/ID.IR 16..31;  |
| **EX** | **ALU**  
        | EX/MEM.IR ← ID/EX.IR;  
        | EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B;  
        | Or  
        | EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;  
        | EX/MEM.cond ← 0;  
        | **Load or Store**  
        | EX/MEM.IR ← ID/EX.IR;  
        | EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;  
        | **Branch**  
        | EX/MEM.ALUOutput ← ID/EX.NPC + ID/EX.Imm;  
| **MEM** | MEM/WB.IR ← EX/MEM.IR;  
           | MEM/WB.ALUOutput ← EX/MEM.ALUOutput;  
           | MEM/WB.IR ← EX/MEM.IR;  
           | MEM/WB.LMD ← Mem[EX/MEM.ALUOutput];  
           | Or  
           | Mem[EX/MEM.ALUOutput] ← EX/MEM.B ;  
| **WB**  | Regs[MEM/WB.IR 16..20] ← EM/WB.ALUOutput;  
        | For load only:  
        | Regs[MEM/WB.IR 11..15] ← MEM/WB.LMD;  
        | Or  
        | Regs[MEM/WB.IR 11..15] ← MEM/WB.ALUOutput;  

---
Pipeline Hazards

• Cases that affect instruction execution semantics and thus need to be detected and corrected

• Hazards types
  – Structural hazard: attempt to use a resource two different ways at same time
    • Single memory for instruction and data
  – Data hazard: attempt to use item before it is ready
    • Instruction depends on result of prior instruction still in the pipeline
  – Control hazard: attempt to make a decision before condition is evaluated
    • branch instructions

• Hazards can always be resolved by waiting
Visualizing Pipelining

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Instr. Order

Slide: David Culler
Example: One Memory Port/
Structural Hazard

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Load
Instr 1
Instr 2
Instr 3
Instr 4

Structural Hazard

Slide: David Culler
Resolving Structural Hazards

1. Wait
   – Must detect the hazard
     • Easier with uniform ISA
   – Must have mechanism to stall
     • Easier with uniform pipeline organization

2. Throw more hardware at the problem
   – Use instruction & data cache rather than direct access to memory
Pipelining Speedup = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}
= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

\text{Ideal CPI pipelined} = 1

\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}
= 1 + \text{Pipeline stall cycles per instruction}

\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

\text{Assuming all pipeline stages are balanced}

\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}