Instruction Set Architecture: Critical Interface

- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Example: MIPS

Programmable storage

<table>
<thead>
<tr>
<th>Format</th>
<th>Data types?</th>
<th>Addressing Modes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit x bytes</td>
<td>31 x 32-bit GPRs (R0=0)</td>
<td>32 x 32-bit FP regs (paired FP)</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td></td>
</tr>
</tbody>
</table>

Arithmetic logical
- Add, AddI, Sub, SubI, And, Or, Xor, SLT, SLTI, AddI, AddIU, SLTI, SLTIU, AndI, Ori, Xori, SLL, SRL, SRA

Memory Access
- LB, LBU, LH, LHU, LW, SB, SH, SW

Control
- JAL, JR, JALR
- BEq, BNE, BLTZ, BGTZ, BLTZ, BGEZ
- 32-bit instructions on word boundary

Instruction Set Architecture

"...the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."

- Amdahl, Blaauw, and Brooks, 1964

- Old definition of computer architecture = instruction set design
  - Other aspects of computer design called implementation
  - Insinuates implementation is uninteresting or less challenging
- H&P’s view is computer architecture >> ISA
- Architect’s job much more than instruction set design; technical hurdles today more challenging than those in instruction set design
- Since instruction set design not where action is, some conclude computer architecture (using old definition) is not where action is
  - H&P disagree on conclusion
  - Agree that ISA not where action is (ISA in CA:AQA 4/e appendix)
Comp. Arch. is an Integrated Approach

- What really matters is the functioning of the complete system
  - hardware, runtime system, compiler, operating system, and application
  - In networking, this is called the "End to End argument"
- Computer architecture is not just about transistors, individual instructions, or particular implementations
  - E.g., Original RISC projects replaced complex instructions with a compiler + simple instructions

A "Typical" RISC ISA

- 32-bit fixed format instruction (4 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

Example: MIPS

\[
\begin{array}{ccccccc}
31 & 26 & 25 & 24 & 23 & 22 & 21 \\
\text{rt} & \text{op} & \text{rd} & \text{rt} & \text{rs} & \text{OP} & \text{rt} \\
\hline
31 & 26 & 25 & 24 & 23 & 22 & 21 \\
\text{rt} & \text{op} & \text{rt} & \text{OP} & \text{immed} & \text{rt} & \text{OP} \\
\hline
31 & 26 & 25 & 24 & 23 & 22 & 21 \\
\text{op} & \text{target} & \text{rt}
\end{array}
\]

5 Steps of MIPS Datapath

- Instruction Fetch
- Instr. Decode
- Execute Addr. Calc
- Memory Access
- Write Back

Example MIPS instructions:

- \[ \text{add} \quad \text{rt}, \text{rs}, \text{rd} \]
- \[ \text{mflo} \quad \text{rt} \]
- \[ \text{mul} \quad \text{rt}, \text{rt}, \text{rs} \]

Example MIPS Datapath Diagram

- Instruction Memory
- Instruction Register
- Decode
- Register Files
- ALU
- Memory
- Write Buffer

Example MIPS instructions:

- \[ \text{add} \quad \text{rt}, \text{rs}, \text{rd} \]
- \[ \text{mflo} \quad \text{rt} \]
- \[ \text{mul} \quad \text{rt}, \text{rt}, \text{rs} \]
5 Steps of MIPS Datapath

June 12, 1989

**Figure A.18, Page A-31**

**Inst. Set Processor Controller**

- **Figure A.2, Page A-8**

**Visualizing Pipelining**

- **Figure A.4, Page A-14**

**One Memory Port/Structural Hazards**

- **Figure A.4, Page A-14**

**Pipelining is not quite that easy!**

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - **Structural hazards:** HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

**How do you “bubble” the pipe?**
**Speed Up Equation for Pipelining**

\[
CPI_{\text{pipeline}} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipeline}}}{\text{Cycle Time}_{\text{pipeline}}}
\]

**Example: Dual-port vs. Single-port**

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed
  
  \[
  \text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1} \times \frac{\text{Clock unpipe}}{\text{Clock pipe}}
  \]
  
  \[
  \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1.4} \times \frac{\text{Clock unpipe}}{1.05}
  \]
  
  \[
  = \frac{\text{Pipeline Depth}}{1.4} \times 1.05
  \]
  
  \[
  = 0.75 \times \text{Pipeline Depth}
  \]
  
- Machine A is 1.33 times faster

**Data Hazard on R1**

<table>
<thead>
<tr>
<th>Instr Order</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Three Generic Data Hazards**

- **Read After Write (RAW)**
  Instr, tries to read operand before Instr, writes it

\[
I: \text{add } r1, r2, r3
\]

\[
J: \text{sub } r4, r1, r3
\]

- Caused by a "true / flow dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

- **Write After Read (WAR)**
  Instr, writes operand before Instr, reads it

\[
I: \text{sub } r4, r1, r3
\]

\[
J: \text{add } r1, r2, r3
\]

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

- **Write After Write (WAW)**
  Instr, writes operand before Instr, writes it.

\[
I: \text{sub } r1, r4, r3
\]

\[
J: \text{add } r1, r2, r3
\]

- Called an "output dependence" by compiler writers. This also results from the reuse of name "r1".

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in more complicated pipes
Forwarding to Avoid Data Hazard

Figure A.7, Page A-18

```plaintext
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-29

```plaintext
add r1, r2, r3
lw r4, 0(r1)
sw r4, 12(r1)
or r8, r6, r9
xor r10, r9, r11
```

Data Hazard Even with Forwarding

Figure A.9, Page A-20

```plaintext
lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
```

Software Scheduling Instead

Try producing fast code for

```
a = b + c;
d = e - f;
```

assuming a, b, c, d, e, and f in memory.

Slow code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
SW a,Re
LW Rf,f
SUB Rb,Re,Rf
SW d,Rd
```

Fast code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
SW a,Re
LW Rf,f
SUB Rb,Re,Rf
SW d,Rd
```

Compiler optimizes for performance, hardware checks for safety.