Introduction

RISC vs CISC

- Introductory lecture → Architectural differences
- How does it effect performance?

\[ P = \frac{1}{I \times C \times S} \]

where

- \( P \) = Performance measure
- \( I \) = number of instructions executed
- \( C \) = clock cycles per instruction
- \( S \) = clock speed

- RISC attempts to reduce \( C \)
- CISC attempts to reduce \( I \)
- A RISC will execute around 10-30% more instructions that CISC
Introduction

RISC – I

RISC I – Register Windows

Processor Data Path

Pipelining

ISA Hazards

History

- First RISC → IBM 801 [1980]
- MIPS → 1981 (J. L. Hennessy at Stanford)
- RISC-I → 1982 (D. Patterson at Berkeley, commercialized as SPARC)
- Currently ARM the most popular RISC machine
- RISC-V gaining importance due to being open source

Notes

Various RISC Machines

<table>
<thead>
<tr>
<th></th>
<th>Alpha</th>
<th>MIPS I</th>
<th>PA-RISC 1.3</th>
<th>PowerPC</th>
<th>SPARC u/c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction size (bits)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Address space (virtual, model)</td>
<td>64 bits, flat</td>
<td>32 bits, flat</td>
<td>48 bits, segmented</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Unaligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Protection</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
</tr>
<tr>
<td>Minimum page size</td>
<td>64 KB</td>
<td>4 KB</td>
<td>4 KB</td>
<td>4 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>MD</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
<tr>
<td>Integer registers (number, model, size)</td>
<td>32 GPR</td>
<td>31 GPR</td>
<td>31 GPR</td>
<td>32 GPR</td>
<td>31 GPR</td>
</tr>
<tr>
<td>Separate floating point register</td>
<td>32 x 32 or 64 bits</td>
<td>56 x 32 or 64 bits</td>
<td>32 x 32 or 64 bits</td>
<td>32 x 32 or 64 bits</td>
<td>32 x 32 or 64 bits</td>
</tr>
<tr>
<td>Floating-point format</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
</tr>
</tbody>
</table>

Figure R.1. Summary of the first version of five recent architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure R.3A. Later versions of these architectures all support a flat, 64-bit address space.

Various RISC Machines

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>Thumb</th>
<th>MIPS6</th>
<th>MIPS16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction size (bits)</td>
<td>32</td>
<td>16</td>
<td>16</td>
<td>16/32</td>
</tr>
<tr>
<td>Address space (virtual, model)</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Integer registers (number, model, size)</td>
<td>15 GPR x 32 bits</td>
<td>8 GPR x 32 bits</td>
<td>16 GPR x 32 bits</td>
<td>16 GPR x 32 bits</td>
</tr>
<tr>
<td>I/O</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>

Figure R.2. Summary of five recent architectures for embedded applications. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure R.3A.
This Session

- This session:
  - RISC-I architecture (5 contact hours)
  - RISC-I pipeline DLX/MIPS processor pipelining (4 contact hours)

Introduction

- Designed by MSc students under D. Patterson and C. H. Sequin (UCLA Berkeley)
- Released in 1982
- Involved a six-month study phase to evaluate the basic concept
- Support for C and Pascal
- 44,500 transistors
- 2µm process (2000nm)
- Simple design yet higher performance than existing, more complex CISC processors
Historical Importance

UC Berkeley students designed and built the first VLSI reduced instruction-set computer in 1981. The simplified instructions of RISC-I reduced the hardware for instruction decode and control, which enabled a flat 32-bit address space, a large set of registers, and pipelined execution. A good match to C programs and the Unix operating system, RISC-I influenced instruction sets widely used today, including those for game consoles, smartphones and tablets.

Design Philosophy

- Simple design with limited transistors
- Hardware support for most time-consuming events
- Most of functionality of the system through software
- High number of overlapping register windows to reduce the time for expensive load/store instructions

Architecture

- Execution of most instructions in one cycle (excluding the instruction fetch from memory)
- All instructions 32-bits (simplifies instruction decoding) with 31 different instructions
- Load/store architecture
- Simple addressing modes
- 32 32-bit registers for each function and 8 register windows (more on that later)
- A total of 138 registers
- Program counter (PC) and program status word (PSW)
Instruction Format

- Opcode: 128 (theoretical upper limit)
- scc: If set, instruction updates the PSW (indicated by a C in instruction)
- dst: specifies one of 32-registers
- src1: specifies one of 32-registers
- imm, src: if (imm==0) then 5 least significant bits used to specify one of 32-registers else src2 is a sign-extended 13 bit constant
- Y: 19-bit constant primarily used for jumps

Instructions

IA32 Instructions’ Synthesis

How will the following IA32 instructions be synthesized?
- mov Rn, Rm
- cmp Rn, Rm
- test Rn, Rm
- neg Rn
- not Rn
- inc Rn
- mov Rn, N
IA32 Instructions' Synthesis

How will the following IA32 instructions be synthesized?

- `mov Rm, Rn` → `add R0, Rm, Rn`
- `cmp Rm, Rn` → `sub Rn, Rm, R0, {C}`
- `test Rm, Rn` → `and Rn, R0, {C}`
- `mov Rm, 0` → `add R0, R0, Rn`
- `neg Rn` → `sub R0, Rn, Rn`
- `not Rn` → `xor Rn, #0, Rn`
- `inc Rn` → `mov R0, #1, Rn`
- `mov Rm, N` → `mov R0, #N, Rn`

Load/Store Instructions

- 5 load and 3 store instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldl (Rm)S, Rn</code></td>
<td>load 32 long [32 bits]</td>
</tr>
<tr>
<td><code>ldsu (Rm)S, Rn</code></td>
<td>load short unsigned [16 bits]</td>
</tr>
<tr>
<td><code>ldss (Rm)S, Rn</code></td>
<td>load signed [16 bits]</td>
</tr>
<tr>
<td><code>ldbs (Rm)S, Rn</code></td>
<td>load byte unsigned</td>
</tr>
<tr>
<td><code>stl (Rm)S, Rn</code></td>
<td>store long</td>
</tr>
<tr>
<td><code>sts (Rm)S, Rn</code></td>
<td>store short [low 16 bits of register]</td>
</tr>
<tr>
<td><code>stb (Rm)S, Rn</code></td>
<td>store byte [low 8 bits of register]</td>
</tr>
</tbody>
</table>

- Indexed addressing `[Rm+ S]`
- `S` must be constant

Addressing Modes

- Register → `add R0, Rm, Rn`
- Immediate → `add R0, #N, Rn`
- Indexed → `ldl (Rm)S, Rn`
- Absolute/direct → `ldl (R0)S, Rn`
- Register indirect → `ldl (Rm)0, Rn`
**Introduction**

- RISC I: A reduced instruction set VLSI computer
- Design and Implementation of RISC I

**Notes**

- **RISC I – Register Windows**

**Procedure Call/Return**

- RISC-I Pre-study: Procedure call/return most time consuming
- RISC in general: Use of registers for passing arguments
- Nested function call?
  - Dump registers on stack
  - Complex instructions implemented as sub-routines
Solution?

- Large register file: 138
- Is that enough?
- Should we assign all these registers to one function?
- How do we pass arguments? Move values between registers?
- What about global values? Memory?

Single Register Window

- One function only sees 32-registers

<table>
<thead>
<tr>
<th>Register Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>r26, r31</td>
</tr>
<tr>
<td>for parameters passed to this function</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>r16, r25</td>
</tr>
<tr>
<td>for local variables &amp; intermediate results</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>r10, r15</td>
</tr>
<tr>
<td>parameters for next function</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>r9</td>
</tr>
<tr>
<td>global registers common to all functions</td>
</tr>
</tbody>
</table>

Overlapped Register Windows

- Overlapped register file simplifies argument passing
Keeping Track

- CWP (current window pointer) keeps track
- Function call → CWP updated
- Caller R10 ... R15 mapped to Callee R26 ... R31

Procedure Call/Return

**CALL** instruction

```plaintext
call (Rsrc1+S2, Rds) ; Call indexed
CWP = CWP+1 ; move to next register window
Rdst = PC ; return address
PC = Rsrc1+S2 ; function start address
```

**CALLR** instruction

```plaintext
call Rdst, Y ; Call relative
CWP = CWP+1 ; move to next register window
Rdst = PC ; return address
PC = PC+Y ; function start address
```

**RET** instruction

```plaintext
RET (Rdst)+S2 ; return
PC = Rdst+S2 ; return address
CWP = CWP-1 ; previous register window
```

- CALL/CALLR and RET must use same Rdst
- In most cases, single cycle function call
Register Window Overflow/Underflow

- What happens if function nest too deeply? What is there are no more register windows to allocate?
- Referred to as Overflow

![Image](http://alanclements.org/register%20windows.html)

**FIGURE 8.32** Change in procedure nesting depth over time. The boxes show procedure calls and returns inside the buffer before a window overflow or underflow. The program starts with three calls, a return, a call, a return, three calls, and then a window overflow.

- Need a mechanism for overflow/underflow handling

Overflow vs Register Windows

![Image](http://alanclements.org/register%20windows.html)

**FIGURE 8.33** Number of banks or windows of registers versus overflow rate for several programs in C, Lisp, and Smalltalk. The programs measured for C include a C compiler, a Pascal interpreter, and a sort program, and a few UNIX utilities (Melton and Reesler 1980). The Lisp measurements include a circuit calculator, a theorem prover, and a functional Lisp interpreter (Tulip et al. 1984). The Smalltalk programs consist of the Smalltalk-80 system (Baker et al. 1983) which includes a compiler, browser, and text editor (Blackman 1982) and Urgo 1982).

Circular Register File

- Register overflow → Need to create space in the register file for a new register window
- Save the oldest register window onto the stack
- Need to maintain the caller/callee relationship
- 8 register windows → 7/8 active windows

Image obtained from: [http://alanclements.org/register%20windows.html](http://alanclements.org/register%20windows.html)
Handling Overflow

- **SWP** → save window pointer (points to the oldest window)
- **CWP++** and **SWP++** (modulo arithmetic)
- Theoretical penalty of 40 clock cycles (280 cycles in SPARC)

Before executing CALL/CALLR:

```c
if (WUSED == NWINDOWS)
    overflowTrapHandler();
    SWP++;
else if (WUSED++)
```
Handling Underflow

Before executing RET:

```c
if (WUSED == 2) {
    SWP--;  // always need 2 valid register windows
    underflowTrapHandler();
} else {
    WUSED--;
}
CWP--;  // and SWP--
```

Notes:

- How does one allocate variables to registers that are accessed through a pointer?
- Solution 1 → Allow compiler to determine variables that have pointers and put them in memory (Slow)
- Solution 2 → Reserve a portion of address space and assign them to registers
  - Determine with one comparison whether address points to register or memory

Pointers to Variables
Learning Goals?

- Processor Pipelining
- Benefits
- Complexities

Single Cycle Processor – Instruction Cycle

William Stallings
One Cycle – One Instruction

- Only start the next instruction cycle when the current cycle is complete.
- Single cycle implementation.

Clock

\[ \text{FI DI CO FO EI Wr} \]

How long is the clock cycle?

- Determined by the critical path.
- Critical path → Delay between two registers or input/output.

Need to Improve the Longest Path
**Longest Path in Processor**

- Determined by the instruction that involves most operation or takes most time to complete

**Multi Cycle Implementation**

- Same operation → Multiple small cycles to complete
- Breaking down the critical path into smaller paths
- Insert registers in between
- Allows multiple operations to be overlapped
- Improves execution time and throughput while latency remains same

**Multiple Cycles – Overlapped Instruction**

- Execute each step within an instruction cycle in one cycle → A pipeline stage
- Overlap the execution of instruction cycles by initiating an instruction cycle as soon as the previous instruction moves to the 2nd stage
- Pipelined multi-cycle implementation → Pipelined implementation
Pipelining – Parallel Execution

- Almost parallel
- Assuming sequential execution

Clock

Pipelining – Datapath Modification

RISC-I Pipelining

- two stage pipeline - fetch unit and execute unit
- normal instructions
  
<table>
<thead>
<tr>
<th>fetch 1</th>
<th>execute 1</th>
<th>fetch 2</th>
<th>execute 2</th>
<th>fetch 3</th>
<th>execute 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch 1</td>
<td>fetch 1</td>
<td>fetch 2</td>
<td>fetch 2</td>
<td>fetch 3</td>
<td>fetch 3</td>
</tr>
<tr>
<td>mem</td>
<td>addr</td>
<td>mem</td>
<td>addr</td>
<td>mem</td>
<td>addr</td>
</tr>
<tr>
<td>load</td>
<td>store</td>
<td>load</td>
<td>store</td>
<td>load</td>
<td>store</td>
</tr>
<tr>
<td>mem</td>
<td>addr</td>
<td>mem</td>
<td>addr</td>
<td>mem</td>
<td>addr</td>
</tr>
<tr>
<td>load</td>
<td>store</td>
<td>load</td>
<td>store</td>
<td>load</td>
<td>store</td>
</tr>
</tbody>
</table>

- load/store instructions
  
<table>
<thead>
<tr>
<th>fetch load</th>
<th>compute addr</th>
<th>mem access</th>
<th>execute load</th>
<th>fetch load</th>
<th>compute addr</th>
<th>mem access</th>
<th>execute load</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
<td>fetch load</td>
</tr>
</tbody>
</table>
MIPS Pipelining

**Notes**

### MIPS Pipelining

3 Hennessy & Patterson

**Notes**

#### Analogy to a Real Life Problem

<table>
<thead>
<tr>
<th>Time</th>
<th>Task order</th>
<th>Task</th>
<th>Task</th>
<th>Task</th>
<th>Task</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>7</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 AM</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
With $n$ instructions and $k$ pipeline stages, total time to execute instructions:
- Non-pipelined $\rightarrow T = nk\tau$ where $\tau \rightarrow$ cycle time period
- Pipelined
  - Initial instruction takes $k$ cycles
  - Every other instruction takes an extra cycle
  - $T = k\tau + (n-1)\tau$

MIPS/DLX Instruction Set Architecture (ISA)

- 32 internal registers for integers and 32 internal registers for floating-point numbers
- Each register is 32-bit wide
- Byte addressable memory $\rightarrow$ A full/half word is also accessible (1 word = 4 bytes)

<table>
<thead>
<tr>
<th>Name</th>
<th>Register #</th>
<th>Usage</th>
<th>Preserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero$</td>
<td>0</td>
<td>the constant value 0</td>
<td>N.A.</td>
</tr>
<tr>
<td>$at$</td>
<td>1</td>
<td>reserved for the assembler</td>
<td>N.A.</td>
</tr>
<tr>
<td>$v0$–$v1$</td>
<td>2-3</td>
<td>value for results and expressions</td>
<td>N</td>
</tr>
<tr>
<td>$a0$–$a3$</td>
<td>4-7</td>
<td>arguments (procedures/functions)</td>
<td>Y</td>
</tr>
<tr>
<td>$s0$–$s7$</td>
<td>8-15</td>
<td>temporaries</td>
<td>N</td>
</tr>
<tr>
<td>$t0$–$t7$</td>
<td>16-23</td>
<td>saved</td>
<td>Y</td>
</tr>
<tr>
<td>$t8$–$t9$</td>
<td>24-25</td>
<td>more temps</td>
<td>N</td>
</tr>
<tr>
<td>$k0$–$k1$</td>
<td>26-27</td>
<td>reserved for the operating system</td>
<td>N.A.</td>
</tr>
<tr>
<td>$g0$</td>
<td>28</td>
<td>global pointer</td>
<td>Y</td>
</tr>
<tr>
<td>$sp$</td>
<td>29</td>
<td>stack pointer</td>
<td>Y</td>
</tr>
<tr>
<td>$fp$</td>
<td>30</td>
<td>frame pointer</td>
<td>Y</td>
</tr>
<tr>
<td>$ra$</td>
<td>31</td>
<td>return address</td>
<td>Y</td>
</tr>
</tbody>
</table>
MIPS Instruction Length

- Length: 32-bits
- Encoding: 3 formats, simple encoding

http://max.cs.kazo.edu/cse230/Resources/MIPS/MachineXL/InstructionFormats.html

Some Instructions

- Data processing
  
  ```
  add $t2,$t0,$t1
  and $t2,$t0,$t1
  srl $t2,$t0, 3
  sra $t2,$t0, 3
  ```

- Data transfer
  
  ```
  lw $t0,32($s3)
  ```

Some Examples

- Data movement within registers
  
  ```
  li $t0, 42
  move $t1, $t0
  ```

- Control instructions later
Immediate – MIPS Example

- Immediate → The operand contains data
  1. addi $sp,$sp,-12 (16 bit imm. sign extended)
  2. addi $s1,$s2,100 (16 bit imm. sign extended)
  3. or $s1,$s2,100 (16 bit imm. zero appended)
  4. lui $s1,100 (16 bit imm. zero extended)
  5. ...

Register Direct – MIPS Example

- Register direct → Operand contains address of the register
  1. addu $s2,$s1,$s3
  2. mfhi $t2
     | The output of mult $t1,$t2 is stored in special registers $t1,$t2
     | mfhi used to move the most significant 32 bits of the 64 bit result in the destination register
     | ...

Direct Addressing – MIPS Example

- Direct → The operand contains the address of a memory location
  MIPS implements a slight variant, referred to as pseudo-direct addressing
  Example
  1. j <label>
     | label is a 26 bit address of a memory location, while we need a 32 bit address
     | PC ← {PC[2:0], label:00} (Concatenation)
     | Why do we concatenate 00 in the end?
     | MIPS memory is byte addressable
     | Instructions are 32 bits, or 4 bytes long
     | There are 2^32 bytes or 2^30 words addressable
     | Only need 30 bits to address a whole word
Register Indirect Addressing – MIPS Example

- Register indirect → The operand points to the register → The contents of the register points to a memory address
- Can specify any address in memory as register is 32-bits
- `jr $ra` → Jump to address stored in register $ra ($31)
- Typically used for returning from functions or procedures
- The last two bits must be 00

PC-Relative Addressing – MIPS Example

- An example of displacement addressing → Can be a combination of immediate, direct and register indirect addressing, i.e., contents of various fields are added to produce the final address
- Used in branching instructions in MIPS
- PC relative means the effective address will be relative to the PC value
- Actually it is the next PC value (PC+4) as PC is incremented as the current instruction is being fetched

Example
- `beq $0, $3, 25`
- The effective address will be:
  - `PC ← PC + 4 + sign-extended(25 << 2)` (in other words, append 00 to the label)
  - `PC ← PC+4 + 100`
- Why PC relative?
  - Conditional branches generally used to implement loops and if-else type conditions
  - Part of the same program so need to jump to nearby locations
  - Effectively done by adding some number to the current PC value
### PC-Relative Addressing – MIPS Example

At the time of fetching/executing the branch instruction:
- $PC + 4 = 0 \times 40000010$
- What shall be the label?
- Difference between $0 \times 40000010$ and $0 \times 40000018$ is $0 \times 8$
- $0 \times 8 = 1000$
- Label is shifted left by 2 bits to make it **word aligned**
- Label ← $0 \times 2(0010)$ stored as 16 bit → $0 \times 00000002$
- Labels can be negative

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \times 40000008$</td>
<td>addi $t2, t2, 100$</td>
</tr>
<tr>
<td>$0 \times 4000000C$</td>
<td>beq $zero, t2, label$</td>
</tr>
<tr>
<td>$0 \times 40000010$</td>
<td>addi $t3, t4, 200$</td>
</tr>
<tr>
<td>$0 \times 40000014$</td>
<td>label addi $t3, t4, 200$</td>
</tr>
<tr>
<td>$0 \times 40000018$</td>
<td>label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000150</td>
<td>00000</td>
<td>01010</td>
<td>0000000000000010</td>
</tr>
</tbody>
</table>

### Base Addressing

- Another example of displacement addressing
- Address of an operand is the sum of the immediate and the value in a register
- Immediate is signed
- Example:
  - lw $t0, 32($s3)
  - Effective address = $32 +$ contents of $s3$
Pipelining Hazards

No Free Lunch

- Pipeline improves performance
- Introduces new complexities → Hazards
- Hazards → Conflict between pipeline stages
- Three types
  - Structural Hazard
  - Data Hazard
  - Control Hazard

Structural Hazard

- Also referred as Resource Hazard
- What does it mean?
- Multiple instructions accessing the same resource
- Example
  - Instruction fetching and operand fetching in the same cycle
  - Solution → Use split cache (Harvard architecture)
RISC-I Pipeline

- two stage pipeline: fetch unit and execute unit
- normal instructions:
  - fetch I
  - execute I
  - fetch I2
  - execute I2
- load/store instructions:
  - fetch load
  - compute addr
  - mem access
  - stall
  - execute 2
  - fetch I3

MIPS Pipelining

Assume the following two MIPS instructions:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Looking at it from a MIPS perspective (5 stage pipeline):
- Sequential execution → No problem
- Pipelined execution → Old data read, termed as a read-after-write (RAW) hazard

Data Hazard

- Assume the following two MIPS instructions:
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

- Looking at it from a MIPS perspective (5 stage pipeline)
- Sequential execution → No problem
- Pipelined execution → Old data read, termed as a read-after-write (RAW) hazard
ALU Interlock

Data Hazard – Solution?
- What can be the possible solution?
- Delay the second instruction
- Titled either as inserting a *nop (no operation)* instruction or a bubble (H&P)
- Slows down the execution

Data Hazard – Another Good Solution
- What can it be?
- When is the data available?
- Alter ALU
- Can we do something?
- Instead of waiting for data to be written to register, *forward* the result to the execution stage → Will entail use of multiplexers and control logic for implementation

```
add $s0, $t0, $t1
```

Data available here
Data Forwarding – A General Concept

- Data can be forwarded from other stages as well
- Consider the following five instructions
  
  ```
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)
  ```

---

Data Forwarding – Modification to Datapath

---

Notes
Sometimes a NOP (no operation) instruction has to be introduced

**Example**

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

Where in the pipeline will data be available?

Let's look at the pipeline diagram

![Pipeline Diagram](image-url)

Data available after the memory stage
Data cannot be moved back in time
NOP necessary
Data Hazards – Other Types

- This data hazard is called RAW hazard \(\rightarrow\) Read after write
- Other types of data hazards in advanced architectures
  - WAR \(\rightarrow\) Write after read
  - WAW \(\rightarrow\) Write after write
Other Solutions to Data Hazards

- No solution is perfect
- Compiler constructed so as to reorganize instructions to avoid hazards
- Manually write assembly code to avoid it
- Example

\[
A = B+E; \\
C = B+F;
\]

- How will it be implemented in MIPS assuming the three variables are stored in memory and are addressable as offsets from \(t0\)

```assembly
lw $t1, 0($t0)  \\
lw $t2, 4($t0)  \\
add $t3, $t1, $t2  \\
A = B+E;  \\
lw $t4, 8($t0)  \\
add $t5, $t1, $t4  \\
C = B+F;  \\
sw $t5, 12($t0)  \\
sw $t5, 16($t0)
```

Avoiding Data Hazard

- Where are data hazards?

Reorganizing Instructions

```assembly
lw $t1, 0($t0)  \\
lw $t2, 4($t0)  \\
lw $t4, 8($t0)  \\
A = B+E;  \\
add $t3, $t1, $t2  \\
C = B+F;  \\
sw $t3, 12($t0)  \\
add $t5, $t1, $t4  \\
sw $t5, 16($t0)
```
Control Hazard

- Consider the following case:

```
fetch jmp  |  execute jmp  |  fetch iWait  |  execute iWait
```

- One clock cycle → Read regs/ALU/Write regs or Read instruction from memory
- Not possible to calculate destination address and fetch at the same time

RISC-I Solution

- Solution → Delayed Jumps
- A software based solution

```
sub r16, #1, r16 {C}
jne L
sub r17, #1, r17
.
.
.
L: sll r16, 2, r16
```

```
sub r16, #1, r16 {C}
jne L
xor r0, r0, r0 // nop
sub r17, #1, r17
.
.
.
L: sll r16, 2, r16
```
Delayed Jump – An Example

```c
i = 0; // assume i in r16
while (i < j) // assume j in r17
  i += f(i); // parameter in r10,
  // return address saved in r25
  // result returned in r1
k = 0; // assume k in r18
```

Unoptimized translation

- Place `nop [xor r0, r0, r0]` after each `jmp/call/ret`

```assembly
add r0, r0, 16  // i = 0
L0: sub r16, r16, r0 (C) // i < j
  jge L1
  // xor r0, r0, r0 // nop
  add r0, r16, r10 // set up parameter
  // in r10
callr r25, f // return address saved
  // in r25
xor r0, r0, r0 // nop
add r1, r16, r16 // i += f(i)
jmp L0 //
xor r0, r0, r0 // nop
L1: add r0, r0, r18 // k = 0
```

Possible to place useful instructions in the delay slot

```assembly
add r0, r0, 16  // i = 0
L0: sub r16, r16, r0 (C) // i < j
  jge L1
  // add r0, r0, 18 // k can be zeroed
  // many times
  // saved in r25
  // in r25
add r0, r16, r10 // set up parameter
  jmp L0 // in r10
add r1, r16, r16 // i += f(i)
L1:
```
Delayed Jump

Better to execute an instruction in the delay slot
Since the instruction in the delay slot is fetched anyway, might as well execute it
60% of delay slots can be filled with useful instructions [Hennessy & Patterson]

Control Hazard in MIPS

Consider the following
Where do we know the branch address? → Stage 4
Control Hazard

- Hazard? Extra instructions fetched

What to do now?

Solutions?

- Predict the outcome of branching instruction
- Two possible solutions
  - Static → Fixed solution
  - Dynamic → Adaptable solution
Static Branch Prediction

- Predict never taken
  - Continue fetching next instruction after branch assuming branch will fail
  - If branch taken, flush the pipeline → Change each instruction into a nop
  - Effective if branches fail half the time
  - Typical of usual MIPS implementation (specially because of its shorter pipeline)

- Predict always taken
  - Opposite to Predict never taken

- Predict by opcode
  - Branch instruction with certain opcodes more likely to be taken than others

- Choice highly dependent on the application area of the processor

Early Branch Detection

- In predict never taken/always taken → Penalty of a wrong prediction high
- Can be reduced by
  - Early detection of branch decision
  - In MIPS, it is possible to detect branch and calculate its address in the second pipeline stage
Early Branch Detection

- In predict never taken/always taken → Penalty of a wrong prediction high
- Can be reduced by
  - Early detection of branch decision
  - In MIPS, it is possible to detect branch and calculate its address in the second pipeline stage
  - Use delay slots

Static Branch Prediction – Reducing the Penalty

- Allowing one or more instruction(s) after the branch instruction to execute even if branch taken
- If one instruction allowed → One delay slot ...

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xe0000008</td>
<td>addi $t2, $t2, 100</td>
</tr>
<tr>
<td>0xe000000c</td>
<td>beq $zero, $t2, label</td>
</tr>
<tr>
<td>0xe0000010</td>
<td>addi $t3, $t4, 200</td>
</tr>
<tr>
<td>0xe0000014</td>
<td>addi $t3, $t4, 200</td>
</tr>
<tr>
<td>0xe0000018</td>
<td>label addi $t3, $t4, 200</td>
</tr>
</tbody>
</table>

- The instruction(s) after the branch shall be valid and useful
Comparison of Branch Strategies

Assume
- 14% branch instructions
- 65% of branches change PC (branch taken)
- 50% probability of filling branch delay slot with a useful instruction

<table>
<thead>
<tr>
<th>Method</th>
<th>Branch penalty</th>
<th>Effective CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline 3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Stall pipeline 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td></td>
</tr>
</tbody>
</table>

DLX Conditional Branch

DLX Pipeline Operation...

- BNEZ/BEQZ instructions [conditional branch]

```
IF  IR ← M[PC]; PC ← PC+4
ID  A ← R[register]; B ← R[register]; PC1 ← PC; IR1 ← IR
EX  ALU[16] ← PC1 + offset; cond ← R[register] Op 0
MA  if (cond) PC ← ALU[16]
WB  idle
```

Dynamic Branch Prediction

- Predict branch behavior based on past history
- Usually called a branch prediction buffer or branch history table
- A small, fast memory containing a bit that indicates the history

![Dynamic Branch Prediction Diagram]
One Bit Prediction – Drawback?

- A single bit prediction does not work well with loops
- Mispredicts the first and last iterations of a nested loop
- A 2-bit prediction history improves prediction for the first iteration
Control Hazard – Other Solutions?

- Multiple Streams
  - Have two pipelines and pre-fetch each branch into separate pipelines. Flush one depending on decision.
  - Pre-fetch branch target along with the instruction following the branch, keep the pre-fetched instruction until decision made.
- Loop buffer
  - Small and fast memory maintained by fetch stage containing recent instructions.
  - Processor checks the buffer first for branch target.
- 2-level correlating predictor (Core i7, Cortex-A8, Pentiums), tournament predictor (Pentium 4), ...

Pipelined Processors – Some Commercial Examples

- Early Pentiums → 5
- PIII → 10
- P4 → 20 – 31
- Core → 14
- ARM → 3 – 24

Branch Prediction Analysis

- Assume
  - 14% branch instructions
  - 90% probability of branch hitting the branch target buffer [as it has a finite capacity]
  - 90% probability of a correct prediction
  - 1 cycle penalty if branch target buffer needs updating
- Branch penalty calculation
  - Penalty = %hits \times \%mispredictions \times 1 + \%misses \times 1
  - Penalty = (0.9 \times 0.1 \times 1) + (0.1 \times 1)
  - Penalty = 0.19 cycle
  - Branches take 1.19 cycles
  - As 14% branches results in 1.0266 effective clocks per instruction [CPI]
- Compares favourably with delayed branching [1.07]
You are now able to:
- Outline the history of RISCs
- Outline the design criteria and architecture of the RISC-1 microprocessor
- Analyse the operation and limitations of register windows
- Analyse the operation and motivation for delayed jumps
- Develop simple RISC-1 assembly language programs
- Describe the operation of the 5 stage DLX/MIPS pipeline
- Explain how data and load hazards are resolved
- Analyse a number of different approaches for handling control hazards
- Explain how branch prediction is implemented
- Use a DLX/MIPS pipeline simulator
- Predict the number of clock cycles needed to execute DLX/MIPS code segments