Computer Structure

Pipeline

Lihu Rappoport and Adi Yoaz
A Basic Processor

Fetch

- Inst. Cache
  - IP
  - Inst.
  - Cache

Decode

- Register File
  - src1 reg
  - src2 reg
  - dst reg
  - imm
  - opcode

Execute

- ALU
  - src1
  - src2

Memory

- Data Cache
  - address
  - data

Write back

- Mem Rd/Wr
  - ALU source
  - ALU control

Control signals

Write-back value
Pipelined Car Assembly

1 hour 2 hours 1 hour
chassis engine finish

Car 3
Ideal speedup is number of stages in the pipeline. Do we achieve this?
Pipelining

- Pipelining does not reduce the *latency* of single task, it increases the *throughput* of entire workload.

- Potential speedup = Number of pipe stages
  - Pipeline rate is limited by the slowest pipeline stage
    - Partition the pipe to many pipe stages
    - Make the longest pipe stage to be as short as possible
    - Balance the work in the pipe stages

- Pipeline adds overhead (e.g., latches)
  - Time to “fill” pipeline and time to “drain” it reduces speedup
  - Stall for dependencies
    - Too many pipe-stages start to loose performance

- IPC of an ideal pipelined machine is 1
  - Every clock one instruction finishes
Pipelined CPU

Fetch

Decode

Execute

Memory

WB

Instruction Cache

Register File

ALU

Data Cache

Sign Ext.

Control signals

ALU source

ALU Control

Mem Wr

Mem Rd

Instruction

src1

src2

data

data

data

dst

4

+4

dst

dst

dst

dst

+4
Structural Hazard

- Different instructions using the same resource at the same time
- Register File:
  - Accessed in 2 stages:
    - Read during stage 2 (ID)
    - Write during stage 5 (WB)
  - Solution: 2 read ports, 1 write port
- Memory
  - Accessed in 2 stages:
    - Instruction Fetch during stage 1 (IF)
    - Data read/write during stage 4 (MEM)
  - Solution: separate instruction cache and data cache
- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions
Pipeline Example: cycle 1

- **Fetch**: Instruction Cache
- **Decode**: Instruction src1 -> dst, src2
- **Execute**: Register File src1, src2 data
- **Memory**: Data Cache, ALU control
- **WB**: Memory operation

**Example Instructions**:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw</td>
<td>R10,9(R1)</td>
<td>R1</td>
</tr>
<tr>
<td>4</td>
<td>sub</td>
<td>R11,R2,R3</td>
<td>R3</td>
</tr>
<tr>
<td>8</td>
<td>and</td>
<td>R12,R4,R5</td>
<td>R5</td>
</tr>
<tr>
<td>12</td>
<td>or</td>
<td>R13,R6,R7</td>
<td>R7</td>
</tr>
</tbody>
</table>
Pipeline Example: cycle 2

0  lw  R10, 9(R1)
4  sub R11,R2,R3
8  and R12,R4,R5
12 or R13,R6,R7
Pipeline Example: cycle 3

0 lw R10, 9 (R1)
4 sub R11, R2, R3
8 and R12, R4, R5
12 or R13, R6, R7
Pipeline Example: cycle 4

0  lw  R10, 9(R1)
4  sub  R11, R2, R3
8  and  R12, R4, R5
12 or  R13, R6, R7
Pipeline Example: cycle 5

0  lw  R10, 9(R1)
4  sub  R11, R2, R3
8  and  R12, R4, R5
12 or  R13, R6, R7
Program execution order

\[ \text{sub } R2, R1, R3 \]
\[ \text{and } R12, R2, R5 \]
\[ \text{or } R13, R6, R2 \]
\[ \text{add } R14, R2, R2 \]
\[ \text{sw } R15, 100(R2) \]
Using Bypass to Solve RAW Dependency

Program execution order

- `sub R2, R1, R3`
- `and R12, R2, R5`
- `or R13, R6, R2`
- `add R14, R2, R2`
- `sw R15, 100(R2)`

Bypass result directly from EXE output to EXE input
RAW Dependency

Fetch

Decode

Execute

Memory

WB

Instruction Cache

Decoder

Register File

ALU

Data Cache

Memory Read

Memory Write

RAW Dependency

0 lw R4, 9(R1)
4 sub R5, R2, R3
8 and R12, R4, R5
12 or R13, R6, R7
Forwarding Hardware

Fetch

Decode

Execute

Memory

WB

**Instruction Cache**

<table>
<thead>
<tr>
<th>IP 16</th>
<th>lw</th>
<th>R4, 9 (R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>sub R5, R2, R3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>and R12, R4, R5</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>or R13, R6, R7</td>
</tr>
</tbody>
</table>

**Register File**

src1 | data | [R4]

src2 | data | [R5]

**Data Cache**

M[[R1]+9]
Forwarding Control

◆ Forwarding from EXE (L3)
  ✷ if (L3.RegWrite and (L3.dst == L2.src1)) ALUSelA = 1
  ✷ if (L3.RegWrite and (L3.dst == L2.src2)) ALUSelB = 1

◆ Forwarding from MEM (L4)
  ✷ if (L4.RegWrite and
      (((not L3.RegWrite) or (L3.dst ≠ L2.src1)) and
       (L4.dst = L2.src1)) ALUSelA = 2
  ✷ if (L4.RegWrite and
      (((not L3.RegWrite) or (L3.dst ≠ L2.src2)) and
       (L4.dst = L2.src2)) ALUSelB = 2
Register File Split

- Register file is written during first half of the cycle
- Register file is read during second half of the cycle
  \[ \Rightarrow \text{Register file is written before it is read} \Rightarrow \text{returns the correct data} \]
Can't Always Forward

- Load word can still causes a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register

⇒ A hazard detection unit is needed to “stall” the load instruction
Stall If Cannot Forward

if (L2.RegWrite and (L2.opcode == lw) and
( (L2.dst == L1.src1) or (L2.dst == L1.src2) ) ) then stall

◆ De-assert the enable to the L1 latch, and to the IP
  ❖ The dependent instruction (and) stays another cycle in L1
◆ Issue a NOP into the L2 latch (instead of the stalled inst.)
◆ Allow the stalling instruction (lw) to move on

Program execution order

<table>
<thead>
<tr>
<th>Program Execution Order</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw R2, 30 (R1)</td>
<td>1</td>
</tr>
<tr>
<td>and R12, R2, R5</td>
<td>2</td>
</tr>
<tr>
<td>or R13, R6, R2</td>
<td>3</td>
</tr>
<tr>
<td>add R14, R2, R2</td>
<td>4</td>
</tr>
<tr>
<td>sw R15, 100 (R2)</td>
<td>5</td>
</tr>
</tbody>
</table>

Diagram illustrating pipeline stages and data flow.
## Software Scheduling to Avoid Load Hazards

Example: code for (assume all variables are in memory):

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

<table>
<thead>
<tr>
<th>Slow code</th>
<th>Fast code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rb,b</td>
<td>Rb,b</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rc,c</td>
<td>Rc,c</td>
</tr>
<tr>
<td>Stall</td>
<td>Stall</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
</tr>
<tr>
<td>Ra,Rb,Rc</td>
<td>Ra,Rb,Rc</td>
</tr>
<tr>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>a,Ra</td>
<td>a,Ra</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Re,e</td>
<td>Re,e</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Re,e</td>
<td>Re,e</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Re,e</td>
<td>Re,e</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rf,f</td>
<td>Rf,f</td>
</tr>
<tr>
<td>Stall</td>
<td>Stall</td>
</tr>
<tr>
<td>SUB</td>
<td>SUB</td>
</tr>
<tr>
<td>Rd,Re,Rf</td>
<td>Rd,Re,Rf</td>
</tr>
<tr>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>d,Rd</td>
<td>d,Rd</td>
</tr>
</tbody>
</table>

Instruction order can be changed as long as the correctness is kept.
Control Hazards
Control Hazard on Branches

Fetch  Decode  Execute  Memory  WB

0 or
4 jcc 48 (offset=40)
8 and
12 mul
16 sub

jcc target; if cond then IP ← IP + 4 + offset
Control Hazard on Branches

Fetch

Decode

Execute

Memory

WB

IP

Inst. Cache

8

Instruction

src1

src2

dst

Register File

src1 data

src2 data

jcc

+4

Register File

src1

src2

data

ALU

Data Cache

address

data

Sign Ext.

jcc target; if cond then IP ← IP + 4 + offset

0 or
4 jcc 48 (offset=40)
8 and
12 mul
16 sub
Control Hazard on Branches

jcc target; if cond then IP ← IP + 4 + offset

0 or
4 jcc 48 (offset=40)
8 and
12 mul
16 sub
Control Hazard on Branches

jcc target; if cond then IP ← IP + 4 + offset
Control Hazard on Branches

Fetch | Decode | Execute | Memory | WB

Control Hazard on Branches

jcc target; if cond then IP ← IP + 4 + offset

0 or
4 jcc 48 (offset=40)
8 and
12 mul
16 sub
Control Hazard on Branches

The 3 instructions following the branch get into the pipe even if the branch is taken.
## Control Hazard: Stall

- **Stall pipe when branch is encountered until resolved**

- **Stall impact: assumptions**
  - CPI = 1
  - 20% of instructions are branches
  - Stall 3 cycles on every branch

\[
\Rightarrow \text{CPI}_{\text{new}} = 1 + 0.2 \times 3 = 1.6
\]

(CPI\textsubscript{new} = \text{CPI}_{\text{ideal}} + \text{avg. stall cycles / instr.})

We lose 60% of the performance
Control Hazard: Predict Not Taken

- Execute instructions from the fall-through (not-taken) path
  - As if there is no branch
  - If the branch is not-taken (~50%), no penalty is paid

- If branch actually taken
  - Flush the fall-through path instructions before they change the machine state (memory / registers)
  - Fetch the instructions from the correct (taken) path

- Assuming ~50% branches not taken on average
  \[ \text{CPI new} = 1 + (0.2 \times 0.5) \times 3 = 1.3 \]
Dynamic Branch Prediction

- Add a **Branch Target Buffer (BTB)** that predicts (at fetch)
  - Instruction is a branch
  - Branch taken / not-taken
  - Taken branch target

<table>
<thead>
<tr>
<th>Branch IP</th>
<th>Target IP</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- BTB allocated at execute – after all branch info is known
- BTB is looked up at instruction fetch
**BTB**

- **Allocation**
  - Allocate instructions identified as branches (after decode)
    - Both conditional and unconditional branches are allocated
  - Not taken branches need not be allocated
    - BTB miss implicitly predicts not-taken

- **Prediction**
  - BTB lookup is done parallel to IC lookup
  - BTB provides
    - Indication that the instruction is a branch (BTB hits)
    - Branch predicted target
    - Branch predicted direction
    - Branch predicted type (e.g., conditional, unconditional)

- **Update (when branch outcome is known)**
  - Branch target
  - Branch history (taken / not-taken)
BTB (cont.)

- **Wrong prediction**
  - Predict not-taken, actual taken
  - Predict taken, actual not-taken, or actual taken but wrong target

- **In case of wrong prediction – flush the pipeline**
  - Reset latches (same as making all instructions to be NOPs)
  - Select the PC source to be from the correct path
    - Need get the fall-through with the branch
  - Start fetching instruction from correct path

- **Assuming P% correct prediction rate**

  CPI\term\ new = 1 + (0.2 \times (1-P)) \times 3

  - For example, if P=0.7
    CPI\term\ new = 1 + (0.2 \times 0.3) \times 3 = 1.18
Adding a BTB to the Pipeline

Fetch

Decode

Execute

Memory

WB

Look up current IP in I$ and in BTB in parallel

BTB provides predicted target and direction

BTB

50

direct
allocate

Inst. Cache

jcc

0 or 4 jcc 50 8 and ...

50 sub
54 mul
58 add

Register File

src1 data

src2 data

dst

Instruction

src1

src2

Sign Ext.

ALU

Data Cache

address

data

predicted target

Next seq. address

predicted direction

flush and repair

repair target

flush and repair

predicted target

Flush and Repair

predicted direction

Next seq. address

Flush and Repair

predicted target

flush and repair

predicted direction

Next seq. address

Flush and Repair

predicted target

flush and repair

predicted direction

Next seq. address

Flush and Repair

predicted target

flush and repair

predicted direction

Next seq. address

Flush and Repair

predicted target

flush and repair

predicted direction

Next seq. address

Flush and Repair

predicted target

flush and repair

predicted direction

Next seq. address
Adding a BTB to the Pipeline

Fetch
Decode
Execute
Memory
WB

BTB
target
direction
allocate

Inst.
Cache

0 or
4 jcc
and
8 sub

... 50 sub
54 mul
58 add

Register File

Instruction
src1
src2
dst

src1 data
src2 data

Sign Ext.

ALU

Data Cache

address
data

Memory

Flush and Repair

Predicted target

Next seq. address

Predicted direction

Repair target

taking

predicted

target

predicted
direction

allocate

54

4

8

Inst.

Cache

jcc
sub

Instruction

...
Adding a BTB to the Pipeline

- Fetch
- Decode
- Execute
- Memory

- Fetch
- Decode
- Execute
- Memory

- BTB
- Register File
- ALU
- Data Cache

- Instruction
- src1
- src2
- dst
- src1 data
- src2 data
- wt

- Predicted target
- Repair target
- Predicted direction
- Next seq. address
- Take
- Not taken

- Issue flush in case of mismatch
- Verify target (if taken)
- Issue flush in case of mismatch
- Verify direction
Using The BTB

PC moves to next instruction

Inst Mem gets PC and fetches new inst

BTB gets PC and looks it up

IF/ID latch loaded with new inst

Branch ?

yes

BTB Hit ?

yes

Br taken ?

PC ← perd addr

no

PC ← PC + 4

no

IF/ID latch loaded with pred inst

IF/ID latch loaded with seq. inst

EXE

yes

no
Using The BTB (cont.)

1. **Branch?**
   - Yes: Calculate br cond & trgt
   - No: Continue

2. **Update BTB**
   - IF/ID latch loaded with correct inst
   - No: Correct pred?
     - Yes: Continue
     - No: Flush pipe & update PC

**Flowchart Diagram:**
- ID: Branch?
  - Yes: Calculate br cond & trgt
  - No: Continue
- EXE: Update BTB
- MEM: Correct pred?
  - Yes: Continue
  - No: Flush pipe & update PC
- WB: IF/ID latch loaded with correct inst
Backup
## MIPS Instruction Formats

- **R-type** (register insts): 31 26 21 16 11 6 0
  - op | rs | rt | rd | shamt | funct
  - 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

- **I-type** (Load, Store, Branch, inst’s w/imm data): 31 26 21 16 0
  - op | rs | rt | immediate
  - 6 bits 5 bits 5 bits 16 bits

- **J-type** (Jump): 31 26 0
  - op | target address
  - 6 bits 26 bits

**op**: operation 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 the jump instruction
Each memory location
- is 8 bit = 1 byte wide
- has an address

We assume 32 byte address
- An address space of $2^{32}$ bytes

Memory stores both instructions and data
- Each instruction is 32 bit wide
  ⇒ stored in 4 consecutive bytes in memory
- Various data types have different width
Register File

- The Register File holds 32 registers
- Each register is 32 bit wide
- The RF supports parallel
  - reading any two registers and
  - writing any register

- Inputs
  - **Read reg 1/2**: #register whose value will be output on **Read data 1/2**
  - **RegWrite**: write enable
  - **Write reg** (relevant when **RegWrite=1**)
    - #register to which the value in **Write data** is written to
  - **Write data** (relevant when **RegWrite=1**)
    - data written to **Write reg**

- Outputs
  - **Read data 1/2**: data read from **Read reg 1/2**
Memory Components

- **Inputs**
  - **Address**: address of the memory location we wish to access
  - **Read**: read data from location
  - **Write**: write data into location
  - **Write data** (relevant when Write=1) data to be written into specified location

- **Outputs**
  - **Read data** (relevant when Read=1) data read from specified location

Cache

- **Memory components are slow relative to the CPU**
  - A **cache** is a fast memory which contains only small part of the memory
  - **Instruction cache** stores parts of the memory space which hold code
  - **Data Cache** stores parts of the memory space which hold data
The Program Counter (PC)

- Holds the address (in memory) of the next instruction to be executed
- After each instruction, advanced to point to the next instruction
  - If the current instruction is not a taken branch, the next instruction resides right after the current instruction
    \[ PC \leftarrow PC + 4 \]
  - If the current instruction is a taken branch, the next instruction resides at the branch target
    \[ PC \leftarrow \text{target} \quad \text{(absolute jump)} \]
    \[ PC \leftarrow PC + 4 + \text{offset} \times 4 \quad \text{(relative jump)} \]
Instruction Execution Stages

- **Fetch**
  - Fetch instruction pointed by PC from I-Cache

- **Decode**
  - Decode instruction (generate control signals)
  - Fetch operands from register file

- **Execute**
  - For a memory access: calculate effective address
  - For an ALU operation: execute operation in ALU
  - For a branch: calculate condition and target

- **Memory Access**
  - For load: read data from memory
  - For store: write data into memory

- **Write Back**
  - Write result back to register file
  - Update program counter
The MIPS CPU

Instruction fetch
Instruction Decode / register fetch
Execute / address calculation
Memory access
Write back
# Executing an Add Instruction

Add R2, R3, R5 ; R2 ← R3+R5

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>ALU</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 = Add</td>
</tr>
</tbody>
</table>

### Diagram

1. **Add** instruction flow:
   - **PC** + 4
   - **Add** circuit
     - **ALU**
     - **Shift left 2**
     - **Add**
     - **Branch = 0**
     - **MemWrite = 0**
     - **MemtoReg = 0**

2. **Instruction Memory**
   - **Address**
   - **Instruction**
   - **Add Instruction**
   - **RegWrite = 1**
   - **ALUSrc = 0**
   - **aluOp = add**
   - **R3 + R5**
Executing a Load Instruction

\[ \text{LW } R1, (30)R2 \ ; \ R1 \leftarrow \text{Mem}[R2+30] \]

\[
\begin{array}{cccccc}
31 & \text{op} & 26 & \text{rs} & 21 & \text{rt} & 16 & \text{immediate} & 0 \\
\text{LW} & 2 & 1 & & & & 30 & \\
\end{array}
\]
Executing a Store Instruction

\[ \text{SW } R1, (30)R2 \quad ; \quad \text{Mem}[R2+30] \leftarrow R1 \]

<table>
<thead>
<tr>
<th>31</th>
<th>op</th>
<th>26</th>
<th>rs</th>
<th>21</th>
<th>rt</th>
<th>16</th>
<th>immediate</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of executing a store instruction with ALU operations and control flow]
Executing a BEQ Instruction

BEQ R4, R5, 27 ; if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ; else PC ← PC+4

BEQ | 4  | 5  | 27
---|---|---|---
op | 26 | rs | 21 | rt | 16 | immediate | 0
31
## Control Signals

<table>
<thead>
<tr>
<th>func op</th>
<th>10 0000</th>
<th>10 0010</th>
<th>Don’t Care</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
</tr>
</tbody>
</table>

RegDst, ALUSrc, MemtoReg, RegWrite, MemWrite, Branch, Jump are shown in the table above.

The table shows the control signals for different operations. The Don’t Care column indicates the default behavior when the function code is not specified.

Add: Perform addition
Subtract: Perform subtraction
Or: Perform OR operation
Lw: Load Word
Sw: Store Word
Beq: Branch on Equal
Jump: Branch on Jump
Pipelined CPU: Load (cycle 1 – Fetch)

LW  R1, (30)R2 ;  R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>LW</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

IF/ID

PC+4

Instruction

Memory

Add

IF/ID

L2

L3

L4

Instruction

Memory

RegWrite

Read reg 1
Read reg 2
Write reg
Write data

Read data 1
Read data 2

Add

ALUSrc

Zero

result

ALUOp

RegDst

ALUControl

Shift left 2

Add result

Add

ALU

Data

Memory

Write Data

MemRead

MemWrite

Branch

PCSsrc

MemtoReg

Write reg

Write data

[15-15]

[15-0]

[20-16]

[15-11]

Sign extend

[32]

6
Pipelined CPU: Load (cycle 2 – Dec)

LW  R1, (30)R2  ;  R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>2</td>
<td>1</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

IF/ID

Instruction Memory

PC

Address

Instruction

L2

PC+4

ALU

L3

ALUOp

MemWrite

Data Memory

Branch

MemRead

L4

MemtoReg

PCSrc

Control

Write data

ALUSrc

Zero

result

Add

Add result

RegWrite

Write reg

Write data

Read reg 1

Read reg 2

Read data 1

Read data 2

[15-0] 16

Sign extend

[20-16]

[15-11]

[32]

[6]

[6]

[6]
### Pipelined CPU: Load (cycle 3 – Exe)

**Instruction**: LW R1, (30)R2 ; R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>31</td>
</tr>
<tr>
<td>rs</td>
<td>26</td>
</tr>
<tr>
<td>rt</td>
<td>21</td>
</tr>
<tr>
<td>immediate</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

**Hexadecimal Representation**:

```
LW    2   1   30
```

**Diagram**:

[Diagram of the pipeline stages including IF/ID, L2, L3, and L4 with the relevant registers and control signals highlighted.]
Pipeline CPU: Load (cycle 4 – Mem)

LW R1, (30)R2 ; R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>2</td>
<td>1</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
Pipelined CPU: Load (cycle 5 – WB)

LW R1, (30)R2 ; R1 ← Mem[R2+30]

31  op  26  rs  21  rt  16  immediate  0

LW  2  1  30
Datapath with Control
**Multi-Cycle Control**

- Pass control signals along just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Five Execution Steps

- **Instruction Fetch**
  - Use PC to get instruction and put it in the Instruction Register.
  - Increment the PC by 4 and put the result back in the PC.
    
\[
\text{IR} = \text{Memory}[	ext{PC}]; \\
\text{PC} = \text{PC} + 4;
\]

- **Instruction Decode and Register Fetch**
  - Read registers rs and rt
  - Compute the branch address
    
\[
\begin{align*}
A &= \text{Reg}[\text{IR}[25-21]]; \\
B &= \text{Reg}[\text{IR}[20-16]]; \\
\text{ALUOut} &= \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);
\end{align*}
\]
  - We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)
Five Execution Steps (cont.)

• Execution
  ALU is performing one of three functions, based on instruction type:
    ❖ **Memory Reference: effective address calculation.**
      \[ \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) ; \]
    ❖ **R-type:**
      \[ \text{ALUOut} = A \text{ op B} ; \]
    ❖ **Branch:**
      \[ \text{if (A==B) PC = ALUOut} ; \]

• Memory Access or R-type instruction completion

• Write-back step
The Store Instruction

- **sw** rt, rs, imm16

- **mem[PC]**
  - Fetch the instruction from memory

- **Addr <- R[rs] + SignExt(imm16)**
  - Calculate the memory address

- **Mem[Addr] <- R[rt]**
  - Store the register into memory

- **PC <- PC + 4**
  - Calculate the next instruction’s address

**Example:**

\[
\text{Mem[Rs + SignExt[imm16]] <- Rt}
\]
RAW Hazard: SW Solution

• Have compiler avoid hazards by adding NOP instructions

Program execution order

<table>
<thead>
<tr>
<th></th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
<th>CC9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of R2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

- sub R2, R1, R3
- NOP
- NOP
- NOP
- and R12, R2, R5
- or R13, R6, R2
- add R14, R2, R2
- sw R15, 100(R2)

• Problem: this really slows us down!
Delayed Branch

- Define branch to take place AFTER $n$ following instruction
  - HW executes $n$ instructions following the branch regardless of branch is taken or not
- SW puts in the $n$ slots following the branch instructions that need to be executed regardless of branch resolution
  - Instructions that are before the branch instruction, or
  - Instructions from the converged path after the branch
- If cannot find independent instructions, put NOP

Original Code

```
r3 = 23
R4 = R3 + R5
If (r1==r2) goto x
R1 = R4 + R5
X: R7 = R1
```

New Code

```
If (r1==r2) goto x
r3 = 23
R4 = R3 + R5
NOP
R1 = R4 + R5
X: R7 = R1
```
Delayed Branch Performance

- Filling 1 delay slot is easy, 2 is hard, 3 is harder
- Assuming we can effectively fill $d\%$ of the delayed slots

\[
CPI_{\text{new}} = 1 + 0.2 \times (3 \times (1-d))
\]

- For example, for $d=0.5$, we get $CPI_{\text{new}} = 1.3$
- Mixing architecture with micro-arch
  - New generations requires more delay slots
  - Cause computability issues between generations