Computer Structure

Pipeline

Lecturer:
Aharon Kupershtok

Prepared by Lihu Rappoport
A Basic Processor

Fetch

Decode

Execute

Memory

Write back

- PC
- Inst. Cache
- Register File
- Data Cache
- ALU
- Write-back value

Control signals

ALU source

ALU control

Mem Rd/Wr

opcode
decode

src1 reg

src2 reg
dst reg

imm

Inst.

src1

src2
dst

src1 data

src2 data

imm

inst.

Write-back value

write value

mem

data

address

sign ext.

imm

ALU control

Mem Rd/Wr

ALU source

inst.

src1

src2

dst reg

reg

reg

reg

reg

reg

reg

reg

reg

reg

reg

reg
Pipelined Car Assembly

1 hour
chassis

2 hours
engine

1 hour
finish

Car 3
Pipelining Instructions

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
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

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

- `sub R2, R1, R3`
- `and R12, R2, R5`
- `or R13, R6, R2`
- `add R14, R2, R2`
- `sw R15, 100(R2)`
Using Bypass to Solve RAW Dependency

Program execution order

\[
\begin{align*}
\text{sub } & \ R2, \ R1, \ R3 \\
\text{and } & \ R12, \ R2, \ R5 \\
\text{or } & \ R13, \ R6, \ R2 \\
\text{add } & \ R14, \ R2, \ R2 \\
\text{sw } & \ R15, 100(\ R2) \\
\end{align*}
\]

Bypass result directly from EXE output to EXE input
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

L1

PC

Inst. Cache

or

Instruction

L2

decode

Control
signals

Register File

src1

data

src1
data

dst

src2

data

src2

data

dst

[4]

decode

Control
signals

ALU
source

ALU
Control

Memory

L3

sub

Mem Wr

Mem Rd

L4

lw

Data Cache

Mem Wr

Mem Rd

Inst. Cache

L1

Instruction

L2

decode

Control
signals

Register File

src1

data

src1
data

dst

src2

data

src2

data

dst

[4]

L1

L2

L3

L4

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

In
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
⇒ Register file is written before it is read ⇒ returns the correct data

sub R2, R1, R3

and R12, R2, R11
Load word can still cause 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

lw R2, 30(R1)
and R12, R2, R5
or R13, R6, R2
add R14, R2, R2
sw R15, 100(R2)
Software Scheduling to Avoid Load Hazards

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

\[
a = b + c;
\]
\[
d = e - f;
\]

<table>
<thead>
<tr>
<th>Slow code</th>
<th>Fast code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td>LW Rb,b</td>
</tr>
<tr>
<td>LW Rc,c</td>
<td>LW Rc,c</td>
</tr>
<tr>
<td>Stall ADD Ra,Rb,Rc</td>
<td>LW Re,e</td>
</tr>
<tr>
<td>SW a,Ra</td>
<td>ADD Ra,Rb,Rc</td>
</tr>
<tr>
<td>LW Re,e</td>
<td>LW Rf,f</td>
</tr>
<tr>
<td>LW Rf,f</td>
<td>SW a,Ra</td>
</tr>
<tr>
<td>Stall SUB Rd,Re,Rf</td>
<td>SW 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 PC ← PC + 4 + offset
Control Hazard on Branches

Fetch | Decode | Execute | Memory | WB

- PC
- Inst. Cache
- Register File src1, src2, dst
- ALU
- Data Cache

+ 4

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

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

PC

Fetch

Decode

Execute

Memory

WB

Inst. Cache

Instruction

src1

src2

src1 data

src2 data

dst

Register File

ALU

Data Cache

Sign Ext.

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

jcc 48 (offset=40)

0 or

4

8 and

12 mul

16 sub
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 PC ← PC + 4 + offset
Control Hazard on Branches

Fetch | Decode | Execute | Memory | WB

PC 48

Inst. Cache

Instruction

Register File

src1

src2

dst

src1 data

src2 data

ALU

Data Cache

address

data

jcc target; if cond then PC ← PC + 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.

- **Beq**
- **And**
- **mul**
- **sub**

Inst from target
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
    ⇒ CPI_{new} = 1 + 0.2 \times 3 = 1.6

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

We loose 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

- 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**
  \[
  \text{CPI new} = 1 + (0.2 \times (1-P)) \times 3
  \]
  - For example, if P=0.7
    \[
    \text{CPI new} = 1 + (0.2 \times 0.3) \times 3 = 1.18
    \]
Adding a BTB to the Pipeline

Fetch
Decode
Execute
Memory
WB

Flush and Repair
Repair target
Predicted target
Predicted direction
Next seq. address
calc. target

BTB provides predicted target and direction

BTB
Inst. Cache
Register File
ALU
Data Cache

lookup current IP in IC and in BTB in parallel
IC provides the instruction bytes
BTB provides predicted target and direction
flush and repair

0 or 4 jcc 50
8 and
... 50 sub 54 mul 58 add

BTB
register
predict
address
taken

src1 data
src2 data
dst

Instruction
src1
src2
Sign Ext.

src1
dst

Adding a BTB to the Pipeline

Fetch

Decode

Execute

Memory

WB

BTB

target
direction
allocate

Instruction Cache

jcc

Sub

50 taken

54

Register File

src1
data

dst

src2
data

Sign Ext.

ALU

Data Cache

address

data

Flush and Repair

Repair target

Predicted target

Predicted direction

Next seq. address

calc. target
Adding a BTB to the Pipeline

Fetch
Decode
Execute
Memory

BTB

Register File

Instruction Cache

Data Cache

ALU

Flush and Repair

Along with the repair PC

Verify direction

Issue flush in case of mismatch (if taken)

0 or 4 jcc 50
8 and

... 50 sub
54 mul
58 add
Using The BTB

1. **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
   - **BTB Hit?**
     - yes
     - Br taken?
       - yes
       - IF/ID latch loaded with pred inst
       - IF/ID latch loaded with seq. inst
     - no
     - PC ← perd addr
   - no
     - PC ← PC + 4
   - IF/ID latch loaded with new inst
     - Branch?
       - yes
       - IF/ID latch loaded with pred inst
       - IF/ID latch loaded with seq. inst
     - no

Computer Structure 2015 – Pipeline
Using The BTB (cont.)

ID

yes

Branch ?

no

Calculate br cond & trgt

Update BTB

yes

Corect pred ?

no

continue

continue

MEM

WB

Flush pipe & update PC

IF/ID latch loaded with correct inst

continue
Backup
MIPS Instruction Formats

- **R-type** (register insts)
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **I-type** (Load, Store, Branch, inst’s w/imm data)
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **J-type** (Jump)
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- **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
    \[ \text{PC} \leftarrow \text{PC} + 4 \]
  - If the current instruction is a taken branch, the next instruction resides at the branch target
    \[ \text{PC} \leftarrow \text{target} \quad \text{(absolute jump)} \]
    \[ \text{PC} \leftarrow \text{PC} + 4 + \text{offset} \times 4 \quad \text{(relative jump)} \]
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

```
31  op  26  rs  21  rt  16  rd  11  shamt  6  funct  0
| ALU | 3 | 5 | 2 | 0 | 0 = Add |
```

Diagram showing the execution of an Add instruction, illustrating the flow of data and control signals through the ALU and memory operations.
Executing a Load Instruction

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

<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>LW</td>
<td>2</td>
<td>1</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of executing a load instruction](image-url)
Executing a Store Instruction

**SW** R1, (30)R2 ; Mem[R2+30] ← R1

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

**Instruction**: SW

**Immediate**: 30

**Register File**: R2

**ALU**: ALUOp = 0, ALUSrc = 1, ALU res = 0

**Memory**: MemRead = 1, MemWrite = 1, MemtoReg = 0

**Control**: RegWrite = 0, RegDst = 0

**Address**: [PC] + 4

**Data Memory**: Write Data

**alu**: Add, Shift left 2, Zero

**alu control**: ALUOp

**alu src**: ALUSrc

**alu res**: ALU res

**memwrite**: MemWrite = 1

**memtoreg**: MemtoReg = 0

**memread**: MemRead

**sw**: SW

**alu dst**: R2

**alu src 1**: R1

**alu add**: Add

**alu control 1**: ALUOp

**alu control 2**: ALUOp

**alu add 1**: Add

**alu control 2**: ALUOp

**alu add 2**: Add

**alu control 2**: ALUOp

**alu add 3**: Add

**alu control 2**: ALUOp

**alu add 4**: Add

**alu control 2**: ALUOp

**alu add 5**: Add

**alu control 2**: ALUOp

**alu add 6**: Add

**alu control 2**: ALUOp

**alu add 7**: Add

**alu control 2**: ALUOp

**alu add 8**: Add

**alu control 2**: ALUOp

**alu add 9**: Add

**alu control 2**: ALUOp

**alu add 10**: Add

**alu control 2**: ALUOp

**alu add 11**: Add

**alu control 2**: ALUOp

**alu add 12**: Add

**alu control 2**: ALUOp

**alu add 13**: Add

**alu control 2**: ALUOp

**alu add 14**: Add

**alu control 2**: ALUOp
Executing a BEQ Instruction

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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>4</td>
<td>5</td>
<td>27</td>
<td>0</td>
<td>21</td>
<td>16</td>
<td>27</td>
</tr>
</tbody>
</table>
## 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</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>

**Legend:**
- **Add**: Add operation is performed.
- **Subtract**: Subtract operation is performed.
- **Or**: OR operation is performed.
- **lw**: Load Word operation is performed.
- **sw**: Store Word operation is performed.
- **beq**: Branch on Equal operation is performed.
- **jump**: Jump operation is performed.
- **x**: Don’t Care
Pipelined CPU: Load (cycle 1 – Fetch)

LW  R1, (30)R2 ;  R1 ← Mem[R2+30]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>op</td>
<td>26</td>
<td>rs</td>
<td>21</td>
</tr>
<tr>
<td>LW</td>
<td>2</td>
<td>1</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Set**

- **LW** (Load Word)

**Pipeline Stages**

1. **IF/ID** (Instruction Fetch/Instruction Decode)
   - PC (Program Counter) + 4
   - Address from Instruction Memory
   - lw
   - Instruction Decode

2. **IF/ID** (Instruction Decode)
   - ALUSrc (ALU Source)
   - RegWrite (Register Write)
   - Read reg 1
   - Read reg 2
   - Write reg
   - Write data

3. **IF/ID** (Instruction Decode)
   - ALU (Arithmetic Logic Unit)
   - Zero result
   - ALUSrc
   - ALUOp
   - RegDst

4. **IF/ID** (Instruction Decode)
   - Data Memory
   - Address
   - Read Data
   - Memory Read

5. **IF/ID** (Instruction Decode)
   - MemtoReg
   - MemWrite
   - Branch

**Example Instruction**

- LW  R1, (30)R2 ;  R1 ← Mem[R2+30]

**Op Codes**

- **LW** (Load Word)

**Register File**

- [15-0] Sign extend
- [20-16] Extend
- [15-11] RegDst
Pipelined CPU: Load (cycle 2 – Dec)

LW  R1, (30)R2  ;  R1 ← Mem[R2+30]
Pipelined CPU: Load (cycle 3 – Exe)

LW R1, (30)R2 ; R1 ← Mem[R2+30]
Pipelined CPU: Load (cycle 4 – Mem)

LW R1, (30)R2 ; R1 ← Mem[R2+30]

```
31  op  26  rs  21  rt  16  immediate  0
LW  2   1   30
```

LW  R1, (30)R2   ;   R1 ← Mem[R2+30]
Pipelined CPU: Load (cycle 5 – WB)

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>
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.
    
    \[
    IR = \text{Memory}[PC];
    PC = PC + 4;
    \]

- **Instruction Decode and Register Fetch**
  - Read registers rs and rt
  - Compute the branch address
    
    \[
    A = \text{Reg}[\text{IR}[25-21]];
    B = \text{Reg}[\text{IR}[20-16]];
    \text{ALUOut} = PC + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);
    \]
  - 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: sw  rt, rs, imm16
RAW Hazard: SW Solution

- Have compiler avoid hazards by adding NOP instructions

<table>
<thead>
<tr>
<th>Program execution order</th>
<th>Value of R2</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub R2, R1, R3</td>
<td>10</td>
<td>CC1:10</td>
</tr>
<tr>
<td>NOP</td>
<td>10</td>
<td>CC2:10</td>
</tr>
<tr>
<td>NOP</td>
<td>10</td>
<td>CC3:10</td>
</tr>
<tr>
<td>NOP</td>
<td>10</td>
<td>CC4:10</td>
</tr>
<tr>
<td>and R12, R2, R5</td>
<td>10–20</td>
<td>CC5:10–20</td>
</tr>
<tr>
<td>or R13, R6, R2</td>
<td>10–20</td>
<td>CC6:10–20</td>
</tr>
<tr>
<td>add R14, R2, R2</td>
<td>10–20</td>
<td>CC7:10–20</td>
</tr>
<tr>
<td>sw R15,100(R2)</td>
<td>10–20</td>
<td>CC8:10–20</td>
</tr>
<tr>
<td>sw R15,100(R2)</td>
<td>10–20</td>
<td>CC9:10–20</td>
</tr>
</tbody>
</table>

- 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
  \[ \text{CPI}_{\text{new}} = 1 + 0.2 \times (3 \times (1-d)) \]
- For example, for $d=0.5$, we get $\text{CPI}_{\text{new}} = 1.3$
- Mixing architecture with micro-arch
  - New generations requires more delay slots
  - Cause computability issues between generations