Computer Structure

Pipeline

Lecturer:
Roni Kupershtok

Slides prepared by Lihu Rappoport
A Basic Processor

- **Fetch**: PC, Inst. Cache
- **Decode**: Opcode, Control signals
- **Execute**: ALU source, ALU control
- **Memory**: Mem Rd/Write
- **Write back**: Data Cache

**Remark**: Program Counter, PC == IP, Instruction Pointer
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 lose performance

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

Fetch

Decode

Register File

src1

src2

dst

Instruction

Instruction Cache

Decode

Control signals

ALU source

ALU Control

Mem Wr

Mem Rd

Execute

Memory

WB

Data Cache

address

data

ALU

src1 data

src2 data

dst

Sign Ext.

dst

+ 4

PC
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

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

PC | Inst. Cache | Instruction | Instruction | Instruction | Instruction

4 | lw  | R4, 9 (R1) |

0 |

4 | sub | R5, R2, R3 |

8 | and | R12, R4, R5 |

12 | or  | R13, R6, R7 |

Control signals | ALU source | ALU Control | Mem Wr | Mem Rd

alu | sub | lw  |


lw | dst | R5 | dst | R4

Register File

src1 data | src2 data

src1 | src2

dst | dst

sign ext.

0 lw R4, 9 (R1)
Forwarding Hardware

Fetch

Instruction Cache

Decode

Register File

Execute

Memory

WB

L1 ➔ L2 ➔ L3 ➔ L4

Control signals

ALU source

Mem Wr

Mem Rd

lw

PC

16

12

16

4

8

lw R4, 9 (R1)

sub R5, R2, R3

and R12, R4, R5

or R13, R6, R7

Data Cache

M[[R1]+9]

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

Instruction

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
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 **and** 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 PC
  - 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)
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>\text{LW Rb,b}</td>
<td>\text{LW Rb,b}</td>
</tr>
<tr>
<td>\text{LW Rc,c}</td>
<td>\text{LW Rc,c}</td>
</tr>
<tr>
<td>\text{ADD Ra,Rb,Rc}</td>
<td>\text{LW Re,e}</td>
</tr>
<tr>
<td>\text{SW a,Ra}</td>
<td>\text{ADD Ra,Rb,Rc}</td>
</tr>
<tr>
<td>\text{LW Re,e}</td>
<td>\text{LW Rf,f}</td>
</tr>
<tr>
<td>\text{LW Rf,f}</td>
<td>\text{SW a,Ra}</td>
</tr>
<tr>
<td>\text{Stall}</td>
<td>\text{Stall}</td>
</tr>
<tr>
<td>\text{SUB Rd,Re,Rf}</td>
<td>\text{SUB Rd,Re,Rf}</td>
</tr>
<tr>
<td>\text{SW d,Rd}</td>
<td>\text{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

PC → Inst. Cache → Register File
- src1
- src2
- dst

Instruction
- src1 data
- src2 data

ALU
- sign ext.

Data Cache
- address
- data

WB

4 or 50H
jcc target; if cond then PC ← target

0 or 4 jcc 50H
8 and
12 mul
16 sub
Control Hazard on Branches

- **Fetch**
- **Decode**
- **Execute**
- **Memory**
- **WB**

- **PC**: 8
- **Instruction Cache**: 8
- **Register File**: src1, src2, dst
- **Data Cache**: address, data
- **ALU**: src1 data + src2 data
- **Sign Ext.**: src1 data

Operations:
- `0 or`
- `4 jcc 50H`
- `8 and`
- `12 mul`
- `16 sub`
Control Hazard on Branches

Instructions following the branch get into the pipe
Control Hazard on Branches

Instructions following the branch get into the pipe

0 or
4 jcc 50H
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
    \[ \Rightarrow \text{CPI}_{\text{new}} = 1 + 0.2 \times 3 = 1.6 \]

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

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 20% of instructions are branches and ~50% branches not taken on average
  \[ \text{CPI new} = 1 + (0.2 \times 0.5) \times 3 = 1.3 \]
Control Hazard on Branches

If the jcc is resolved as taken at EXE ⇒ flush the pipeline:
Reset the pipe latches, and set PC ← target

0 or
4 jcc 50H
8 and
12 mul
16 sub
Dynamic Branch Prediction

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

![Diagram of BTB](image)

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

 Allocation – at Decode / EXE
  ❖ 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 – at Fetch
  ❖ 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 – at EXE)
  ❖ 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 PC
Predicted target
Predicted direction
Next seq. address

BTB provides predicted target and direction

BTB
Inst. Cache
Register File
ALU
Data Cache

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

Register File
src1 data
src2 data
dst
src1

Data Cache address
data

Sign Ext.

src2

src1

taken

Instruction

4
8

jcc

50

predicted target

predicted direction

Next seq. address

flush and repair

allocate

PC

lookup current PC in I$ and in BTB in parallel
I$ provides the instruction bytes
BTB provides predicted target and direction

flush and repair

predicted direction ≠ predicted direction

flush and repair

flush and repair
Adding a BTB to the Pipeline

Fetch | Decode | Execute | Memory | WB

- BTB target
- Direction allocate
- Predicted target
- Predicted direction
- Next seq. address
- Repair PC
- Flush and Repair
- Register File
- Instruction
- src1
- src1 data
- src2
- src2 data
- dst
- ALU
- Sign Ext.
- Data Cache
- Address
- Data
- sub
- jcc
- or
- and
- mul
- add
Adding a BTB to the Pipeline

Fetch
- BTB
- Target
- Direction
- Allocate

Decode
- Instruction Cache
  - Inst.
- BTB
- Target
- Direction
- Allocate

ALU
- Jcc
- Taken
- 0 or 4
- Jcc
- 50
- 8 and
- ... 50
- Sub
- 54 Mul
- 58 Add

Register File
- Src1
- Data
- Src2
- Data
- Dst

Data Cache
- Address
- Data

Execute
- Repair PC
- Predicted Target
- Predicted Direction
- Next Seq. Address

Memor
- Issue flush in case of mismatch
- Verify target (if taken)
- Verify Direction

Computer Structure 2018 – Pipeline
Along with the repair PC
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

IF/ID latch loaded with pred inst

PC ← pred addr

BTB Hit ?

Br taken ?

IF/ID latch loaded with seq. inst

PC ← PC + 4

Branch ?

yes

no

yes

no

EXE

yes

no
Using The BTB (cont.)

1. **ID**
   - Branch? yes — Calculate br cond & trgt
   - Branch? no — continue

2. **EXE**
   - Update BTB
     - Correct pred? yes — continue
     - Correct pred? no — continue

3. **MEM**
   - continue

4. **WB**
   - IF/ID latch loaded with correct inst
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>
</tbody>
</table>

  - 6 bits
  - 5 bits
  - 5 bits
  - 5 bits
  - 5 bits
  - 6 bits

- **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>
</tbody>
</table>

  - 6 bits
  - 5 bits
  - 5 bits
  - 16 bits

- **J-type** (Jump)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>

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

Add R2, R3, R5 ; R2 ← R3+R5

<table>
<thead>
<tr>
<th></th>
<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>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td>0</td>
</tr>
<tr>
<td>ALU</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0 = Add</td>
<td></td>
</tr>
</tbody>
</table>
Executing a Load Instruction

LW R1, (30)R2 ; R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Add [PC]+4

PC → Address Instruction Memory

Instruction Memory → LW Instruction

LW Instruction → Add

Add → Mem[R2+30]

Mem[R2+30] → ALU Control

ALU Control → ALUOp

ALUOp → Address Data Memory

Address Data Memory → MemRead=1

MemRead=1 → MemtoReg=1

MemtoReg=1 → R2

R2 → ALU

ALU → ALUSrc=1

ALUSrc=1 → Zero Alu

Zero Alu → Branch=0

Branch=0 → MemWrite=0

MemWrite=0 → Mem[MemtoReg=1]

Mem[MemtoReg=1] → Write Data Memory

Write Data Memory → Write Data

Write Data → Read Data Memory

Read Data Memory → Read Data

Read Data → RegWrite=1

RegWrite=1 → Write reg

Write reg → Write reg 2

Write reg 2 → RegDsts=0

RegDsts=0 → [15-0]

[15-0] → [5-0]

[5-0] → [15-11]

[15-11] → [20-16]

[20-16] → [25-21]

[25-21] → [PC]+4

[PC]+4 → Add

Add → 4

4 → Add

Add → [PC]+4

[PC]+4 → PC
Executing a Store Instruction

\[
\text{SW } R1, (30)R2 \ ; \ \text{Mem}[R2+30] \leftarrow R1
\]
Executing a BEQ Instruction

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

<table>
<thead>
<tr>
<th>func op</th>
<th>00 0000 10 0000</th>
<th>00 0010 10 0010</th>
<th>Don’t Care</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>jump</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegDst</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</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 Subtract Or Add Add Subtract xxx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelined CPU: Load (cycle 1 – Fetch)

LW R1, (30)R2 ; R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>
Pipelined CPU: Load (cycle 2 – Dec)

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

\[
\begin{array}{cccccc}
31 & \text{op} & 26 & \text{rs} & 21 & \text{rt} & 16 & \text{immediate} & 0 \\
\hline
\text{LW} & 2 & 1 & 30 & \\
\end{array}
\]
Pipelined CPU: Load (cycle 3 – Exe)

LW  R1, (30)R2  ;  R1 ← Mem[R2+30]

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Pipelined CPU diagram showing the pipeline stages and operations involved in executing the `LW` instruction.
Pipelined CPU: Load (cycle 4 – Mem)

LW R1, (30)R2 ; R1 ← Mem[R2+30]

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

```
LW  R1, (30)R2 ; R1 ← Mem[R2+30]
```

**Diagram:**

- **PC:** Program Counter
- **IF/ID:** Instruction Fetch/Decode
- **L2:** Load Register File
- **L3:** ALU (Arithmetic Logic Unit)
- **L4:** Memory Access
- **MemWrite:** Memory Write
- **MemRead:** Memory Read
- **D:** Data
- **Branch:** Branching control
- **ALUOp:** ALU Operation
- **ALUSrc:** ALU Source
- **RegWrite:** Register Write
- **RegDst:** Register Dst
- **Write reg:** Write register
- **Read reg:** Read register
- **Read data 1:** Read data 1
- **Read data 2:** Read data 2
- **Shift left 2:** Shift left by 2
- **Sign extend:** Sign extend
- **Zero result:** Zero result
- **Add:** Addition
- **Add result:** Add result
- **L:** Instruction
- **mux:** Multiplexer
- **Instruction Memory:** Instruction Memory

**Register File:**

- [15-0] 16
- [20-16]
- [15-11]
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

[Diagram of a datapath with control logic, showing various components such as IF/ID, ID/EX, EX/MEM, MEM/WB, and control signals like PCSrc, Control, Add, ALUSrc, ALUOp, RegWrite, RegDist, ALU, and MemRead.]
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 = Memory[PC];
    PC = PC + 4;

- **Instruction Decode and Register Fetch**
  - Read registers rs and rt
  - Compute the branch address
    
    A = Reg[IR[25–21]];
    B = Reg[IR[20–16]];
    ALUOut = PC + (sign-extend(IR[15–0]) << 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.**
    \[ ALUOut = A + \text{sign-extend}(IR[15-0]); \]
  ❖ **R-type:**
    \[ ALUOut = A \text{ op } B; \]
  ❖ **Branch:**
    \[ \text{if } (A==B) \text{ PC } = \text{ALUOut}; \]

• Memory Access or R-type instruction completion

• Write-back step
## The Store Instruction

### Format

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
</tbody>
</table>

- 6 bits
- 5 bits
- 5 bits
- 16 bits

### Example

- **sw**  
  - **rt**, **rs**, **imm16**

### Execution Steps

1. **Fetch the instruction from memory**
   - `mem[PC]`

2. **Calculate the memory address**
   - `Addr ← R[rs] + SignExt(imm16)`

3. **Store the register into memory**
   - `Mem[Addr] ← R[rt]`

4. **Calculate the next instruction’s address**
   - `PC ← PC + 4`

Example:

```
Mem[Rs + SignExt[imm16]] ← Rt
```

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, CC2 10, CC3 10, CC4 10, CC5 10-20, CC6 -20, CC7 -20, CC8 -20, CC9 -20</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and R12, R2, R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or R13, R6, R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R14, R2, R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw R15, 100(R2)</td>
<td></td>
<td></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**

\[
\begin{align*}
\text{r3} &= 23 \\
\text{R4} &= \text{R3} + \text{R5} \\
\text{If (r1==r2)} &\text{ goto x} \\
\text{R1} &= \text{R4} + \text{R5} \\
\text{X: R7} &= \text{R1}
\end{align*}
\]

**New Code**

\[
\begin{align*}
\text{If (r1==r2)} &\text{ goto x} \\
\text{r3} &= 23 \\
\text{R4} &= \text{R3} + \text{R5} \\
\text{NOP} \\
\text{R1} &= \text{R4} + \text{R5} \\
\text{X: R7} &= \text{R1}
\end{align*}
\]
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