Chapter 4

The Processor
### The Main Control Unit

**Control signals derived from instruction**

<table>
<thead>
<tr>
<th>R-type</th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load/Store</th>
<th>35 or 43</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch</th>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
</tr>
</tbody>
</table>

- **opcode** is always read for R-type and load.
- **write for R-type and load**.
- **sign-extend and add**.
Full Datapath

Chapter 4 — The Processor — 3
ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

### ALU control

<table>
<thead>
<tr>
<th>ALU control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
### ALU Control

- Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>funct</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>
Datapath With Control
R-Type Instruction
Branch-on-Equal Instruction
## Implementing Jumps

<table>
<thead>
<tr>
<th>Jump</th>
<th>[31:26]</th>
<th>[25:0]</th>
</tr>
</thead>
</table>
| Jump uses word address
Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
| Need an extra control signal decoded from opcode |
Datapath With Jumps Added

Chapter 4 — The Processor — 11
Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file

- Not feasible to vary period for different instructions

- Violates design principle
  - Making the common case fast

- We will improve performance by pipelining
Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

### Diagram

- Task order:
  - A
  - B
  - C
  - D

Time line:

- 6 PM
- 7
- 8
- 9
- 10
- 11
- 12
- 1
- 2 AM

- Overlapping execution shown with different stages of each task.
MIPS Pipeline

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
Pipeline Performance

**Single-cycle ($T_c = 800\text{ps}$)**

Program execution order (in instructions)

Instruction fetch | Reg | ALU | Data access | Reg

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$

**Pipelined ($T_c = 200\text{ps}$)**

Program execution order (in instructions)

Instruction fetch | Reg | ALU | Data access | Reg

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions

Load/store addressing
  - Can calculate address in 3\textsuperscript{rd} stage, access memory in 4\textsuperscript{th} stage

Alignment of memory operands
  - Memory access takes only one cycle
Hazards

- Situations that prevent starting the next instruction in the next cycle

  - Structure hazards
    - A required resource is busy

  - Data hazard
    - Need to wait for previous instruction to complete its data read/write

  - Control hazard
    - Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Forwarding (aka Bypassing)

- Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath

Diagram showing the program execution order and time with instructions add $s0, $t0, $t1 and sub $t2, $s0, $t3.
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!

lw $s0, 20($t1)
sub $t2, $s0, $t3
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E; \ C = B + F;$

```
C code for
A = B + E; C = B + F;
```

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

13 cycles
11 cycles
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
Stall on Branch

- Wait until branch outcome determined before fetching next instruction
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
MIPS with Predict Not Taken

Prediction correct

Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40
- lw $3, 300($0)

Prediction incorrect

Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40

or $7, $8, $9
More-Realistic Branch Prediction

- **Static branch prediction**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- **Dynamic branch prediction**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation
Right-to-left flow leads to hazards
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time

- We’ll look at “single-clock-cycle” diagrams for load & store
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load
WB for Load

Chapter 4 — The Processor — 38
Corrected Datapath for Load
EX for Store
MEM for Store
WB for Store
Multi-Cycle Pipeline Diagram

Form showing resource usage

Program execution order (in instructions)

- lw $10, 20($1)
- sub $11, $2, $3
- add $12, $3, $4
- lw $13, 24($1)
- add $14, $5, $6
Multi-Cycle Pipeline Diagram

Traditional form

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC 1</td>
</tr>
<tr>
<td>Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>Instruction decode</td>
<td></td>
</tr>
<tr>
<td>Execution</td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
</tbody>
</table>

lw $10, 20($1)

sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

add $14, $5, $6
Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $14, $5, $6</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>lw $13, 24 ($1)</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>add $12, $3, $4</td>
<td>Execution</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>Memory</td>
</tr>
<tr>
<td>lw $10, 20($1)</td>
<td>Write-back</td>
</tr>
</tbody>
</table>
Pipelined Control (Simplified)
Pipelined Control

- Control signals derived from instruction
- As in single-cycle implementation
Pipelined Control
Data Hazards in ALU Instructions

Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or $13,$6,$2
add $14,$2,$2
sw $15,100($2)
```

We can resolve hazards with forwarding

- How do we detect when to forward?
Dependencies & Forwarding

<table>
<thead>
<tr>
<th>Value of register $2$</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

Program execution order (in instructions)

- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite

- And only if Rd for that instruction is not $zero
  - EX/MEM_REGISTERRd ≠ 0, MEM/WB.RegisterRd ≠ 0
Forwarding Paths

b. With forwarding
Forwarding Conditions

- EX hazard
  - if \((\text{EX/MEM.RegWrite and (EX/MEM.RegisterRd} \neq 0)\)
    and \((\text{EX/MEM.RegisterRd = ID/EX.RegisterRs})\))
    
    ForwardA = 10
  - if \((\text{EX/MEM.RegWrite and (EX/MEM.RegisterRd} \neq 0)\)
    and \((\text{EX/MEM.RegisterRd = ID/EX.RegisterRt})\))
    
    ForwardB = 10

- MEM hazard
  - if \((\text{MEM/WB.RegWrite and (MEM/WB.RegisterRd} \neq 0)\)
    and \((\text{MEM/WB.RegisterRd = ID/EX.RegisterRs})\))
    
    ForwardA = 01
  - if \((\text{MEM/WB.RegWrite and (MEM/WB.RegisterRd} \neq 0)\)
    and \((\text{MEM/WB.RegisterRd = ID/EX.RegisterRt})\))
    
    ForwardB = 01
Double Data Hazard

- Consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Datapath with Forwarding
Load-Use Data Hazard

Program execution order (in instructions)

- lw $2, 20($1)
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2
- slt $1, $6, $7

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9

Need to stall for one cycle
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for \( \text{lw} \)
    - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Program execution order (in instructions)

`lw $2, 20($1)`

and becomes `nop`

and `$4, $2, $5`

or `$8, $2, $6`

`add $9, $4, $2`

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

- `lw $2, 20($1)`
- and becomes `nop`
- and `$4`, `$2`, `$5` stalled in ID
- or `$8`, `$2`, `$6` stalled in IF
- `add `$9`, `$4`, `$2`

Or, more accurately…
Datapath with Hazard Detection
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Branch Hazards

- If branch outcome determined in MEM

Program execution order (in instructions)

- 40 beq $1, $3, 28
- 44 and $12, $2, $5
- 48 or $13, $6, $2
- 52 add $14, $2, $2
- 72 lw $4, 50($7)

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

- Example: branch taken

  36:  sub  $10, $4, $8
  40:  beq  $1,  $3,  7
  44:  and  $12, $2, $5
  48:  or   $13, $2, $6
  52:  add  $14, $4, $2
  56:  slt  $15, $6, $7
        ...
  72:  lw   $4, 50($7)
Example: Branch Taken

and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $8
before<1>
before<2>

Clock 3

Chapter 4 — The Processor — 67
**Example: Branch Taken**

iw $4, 50($7)  

Bubble (nop)  

beq $1, $3, 7  

sub $10, ... before<1>

Clock 4

Chapter 4 — The Processor — 68
Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

- Can resolve using forwarding

```
add $1, $2, $3
add $4, $5, $6
...
beq $1, $4, target
```
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

```
lw $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw   $1, addr
beq   stalled
beq   stalled
beq   $1, $0, target
```