Chapter 4

The Processor

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview

Multiplexers

- Can't just join wires together
  - Use multiplexers

Control

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

- AND-gate
  - Y = A & B

- Multiplexer
  - Y = S ? I1 : I0

- Adder
  - Y = A + B

- Arithmetic/Logic Unit
  - Y = F(A, B)

Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch

Common for all

32-bit register

Increment by 4 for next instruction

Recall: Instruction Formats

shamt — shift operation

Source1 source2 destination

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

add $t1, $t2, $t3 → $t1 = $t2 + $t3

rd rs rt

R format → arithmatic সকল operation cover করে,

I " → · memory (load / store) · immediate · branch } operation কে cover করে

lw $t1, 20($a0) → memory থেকে register এ load

sw $t1, 20($a0)

memory calculation

destination এখানে memory

MIPS এর সব operation এ টার্গেট হয় destination শুরুতে।

ori $t1, $t2, 10

- MIPS এর operation এ branch এর শুধু equality check করতে পারি। greater/less check করা যায় না।
- MIPS word addressing follow করে।

Memory → flat → byte/order
→ byte/word addressible

flat → Banking memory
→ single array র মতো থাকে। linearly.

MIPS এর প্রতিটা memory location এ address করলে 4 byte করে data পাবো।

মাঝখানের একটা address থেকে কোনোভাবেই data নেওয়া যাবে না। ঐ address থেকে next 4 byte data যাবে। — word addressible.

তাই আমরা শুধু প্রথম 2 bit দেই। তাকে left shift করে 2টা 0 add প্রথম address টা পাবে। then সেটার সাথে +1, +2, +3 করবো।

beq $t1, $t2, value

→ 2 flag 1 হবে এবং instruction টা branch হলেই কখন branching operation করবো?

paragraph → যে boundary র নীচে data fetch করতে পারবো না।

এখানে 4 byte paragraph size.

digital hardware design অনুযায়ী —

- combinational — clock লাগে না
- sequential — clock লাগে

Latch	Flip Flop
• clock নাই	• clock আছে

data input দিলেই output পাচ্ছি — combinational part

wait করা লাগলে — sequential

memory → huge bunch of address and data

register file/ register memory → register এর memory

register এর address দিলে সেখানের data পাবো।

RA, RB > read register

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

a. Data memory unit

b. Sign extension unit

6 bits	5 bits	5 bits	5+5+6 = 16 bits
op	rs	rt	addr./imm.

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

Store
Load

6 bits	5 bits	5 bits	5+5+6 = 16 bits
op	rs	rt	addr./imm.

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions

Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

Full Datapath

4 bit control (Control unit থেকে বানাতে হয়।) এই 4bit বানাতে input কি কি দিবো?

6 bit instruction op code
6 bit function

} 12 bit input
4 bit output

→ complex circuit হবে।

– 2 level decoding করবো

1. instruction decoding

00 → lw/sw → add

01 → beq → subtraction

10 → R type → ??

6 bit op code to 2 bit ALU op → 6 to 2

R type ও ALU op 2 bit
function 6 bit
8 bit দেখে decision.

প্রথম 2 bit ignore so → 10 থাকে R type বোঝায়
(2+4) bit নিয়ে decision

ALU op

00 ⟶ funct field ⟶ don't care

01

10

msb 1 হলে R type

<u>function field</u>

প্রথম 2 bit don't care

Assertion — control থেকে এমন ভাবে input পাঠাবো যেন chip 1 হয়।

enable high তে 1

enable low তে 0.

<u>Deassertion:</u>

let যুরোটা active high.

(enable high low matter করে না।)

Reg Dst : 0 হলে IM 16–20 যাবে Rt

1 " 11–15 " Rd

PC Src → derived

কারণ control + 2 bit লাগে।

Mem Read: memory read costly operation. clock cycle বাড়বে

— power consumption বাড়াবে।

সব operation এ লাগে না।

তাই control

electrical device : fully electrical force দিয়ে চলবে

mechanical device : rotate করবে। mechanical energy লাগবে।

Mem Reg: o/p কি memvalue না Alu o/p.

↑ ↑

1 0

Reg Write: write enable (RF এর)

Alu src: immediate নাকি Rt

	Reg dst	ALU Src	Mem to Reg	Reg W	Mem R	Mem W	Branch	ALU op1	ALU op 2
R	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0		0	0	0	1	0	1

jump → register write নাই

ALU source লাগছে না।

— direct address দেওয়া থাকে

ALU Control

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

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

subset of functions

ALU Control

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

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

Datapath With Control

All Control Signals

Signal name	=0 Effect when deasserted	=1 Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See Appendix B for further discussion of this problem.)

R-Type Instruction

Load Instruction

Branch-on-Equal Instruction

Implementing Jumps

Jump	2	address
	31:26	25:0

- 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

Bus-set of wires

1. 28 bit —28টা wire.

তার মানে PC র সব টা wire বসানোই হচ্ছে।

একটা 4x2 mux use.

3টা input.

00 → next

01 → branch

10 → jump

jump, branch একসাথে 1 হবে না।

PC 32 bit হওয়াই লাগবে এমন না।

word index multiplied by 4

→ byte index

word addressing

0 → 0
4 → 1
8 → 2
12 → 3
⋮

word index

— 30 bit এ করে ফেললাম

PC তে 30 bit but IM এ 32 লাগবে।

last এ 2 টা 0 append.

2 টা zero wire append করে দিবো

Full datapath এ কোথায় সবচেয়ে বেশি সময় লাগবে? — critical path.

→ এখানে optimize করা গেলে clock cycle কমে যাবে।

— lw বা sw হবে।

lw এ address আসবে value বের হবে আবার ঘুরবে।

so ans is lw.

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

point A to point B পানি transfer — pipe use

এক ফোঁটা পাঠালাম সেটা অন্যপাশে পৌঁছালে second ফোঁটা ছাড়বো —

এভাবে না। সময় বেশি লাগবে।

throughput time optimized করছি। পুরোটা একসাথে ছেড়ে দেই যেন pipe এর প্রতিটা segment পানি দ্বারা filled থাকে।

1. load carry — প্রতিটা step এ 30 min
2. Washer — full batch 2hr
3. drier — এমন 4টা batch — 8 hr
4. folding

— Resource unutilized থাকছে।

not underutilized.

fill time আর খালি করার time বাদে বাকি পুরোটা time resource full use হবে।

pipeline এর পুরো resource utilize করা যাবে যদি pipeline এর প্রতিটা slice/stage কে equally ভাগ করতে পারি।

n সংখ্যক কাজ —

single path এ 2n

<u>pipeline এ</u>

n সংখ্যক কাজ — 0.5 hr এ done

fill up time 1.5 hr

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 – ALU 를 part
  4. MEM: Access memory operand – memory 를 read/write
  5. WB: Write result back to register

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

8 hr

3.5 hr

resource কে আর full capacity তে use হচ্ছে।

একটা single task এর সমান কাজ 30min এ done – 4 resources occupied

- Four loads:
  - Speedup
    = 8/3.5 = 2.3

maximum speedup না।

- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

speedup = $\frac{800}{200}$

= 4 গুণ

5 stage এ 5 গুণ কেন না?

WB এ 100 waste

ID এ 100 waste

তাই pipelined এর speedup এ কমে যাবে।

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$
    $= \dfrac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to ~~17~~ 15-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

— 8086/ X86 architecture এ pipeline support করানো যায় না। ওদের ISA কে assembly type lang. এ convert then pipeline include করা হয়।

instruction format 3টা, 8 এ 7টা

— mips এ memory কে কোনো arithmatic op এ রাখতে পারি না। x86 এ একটা operand memory হতে পারে।

— mips এর ALU তে memory র কোনো কাজ নেই। ALU র কাজ শেষে memory তে —.

eg: 8086 এ — 16 bit, এক cycle এ বা 2 cycle এ data নিতে পারি।

একটা call দিলে single cycle এ data নেয়ে যাবো এমন নাও হতে পারে।

mips এ এক cycle এই data/instruction পাবো — guaranteed

streamlined জিনিষ লাগবে — RISC ভালো।

mips এ সকল ধরনের write — last এ হয়।

Hazards

→ structural, data, control

- 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 → branching এর সময় হয়
  - Deciding on control action depends on previous instruction

↳ একই slice এ memory র জায়গায় use

— structural hazards

<u>soln:</u> memory ২ ভাগ করি।

— instruction (IM)

— data (DM)

add $1, $2, $3

I1

I2

sub $4, $1, $5

একই slice এ হচ্ছে — সম্ভব না।

Soln: একটা register file এই কাজ করা লাগবে।
আলাদা ২টা rf দিয়ে soln হবে না।

rising এ write করবো

falling এ read করবো (১ম half cycle এ read করা যাবে না)

Structure Hazards

→ একই resource একই সাথে দুই জায়গায় use.

→ একটা single slice (একই time এ হয়) একই resource ২ জায়গায় চাচ্ছে।

- 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

raw hazard → read after write
write করার পরের instruction এ read.
nop instruction — full 32 bit 0।
→ কাজে ২টা ফাঁকা instruction insert করছি। naive soln.

problem: ২টা instruction insert এ ২টা clock cycle বাড়ছে। time waste.

Soln: forwarding দিয়ে solve.

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
    sub $t2, $s0, $t3

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
    sub $t2, $s0, $t3

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)

COMPARE

Load-Use Data Hazard

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

load use data hazard

ready হয় এখানে (forwarding দিয়ে how?) → হবেনা।

→ waiting দিয়ে solve হবে।
waiting is a soln for every hazards.

Load-Use Data Hazard

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

When WAS the data Ready for R-type?

Can we solve by FORWARDING?

When is the data Ready?

Load-Use Data Hazard

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

When WAS the data Ready for R-type?

Can we solve by FORWARDING?

When is the data Ready?

Time Travel!!!

Load-Use Data Hazard

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

Can we solve by FORWARDING?

When is the data Ready?

Load-Use Data Hazard

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

→ code কে restructure করা লাগবে। compiler করে।

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;

raw solve — ২টা stall or forwarding
load use data — ১টা stall লাগে

stall

stall

```
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)
```

13 cycles

```
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)
```

11 cycles

load কে এক ঘর উপরে তুলে দিচ্ছি

```
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)
```

13 cycles

```
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)
```

```
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)
```

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

Branch Scenario

When is the branch decision made?

— 3টা instruction nop করা লাগবে। ভালো soln না।

Branch Taken

top tested loop
— উপরে একটা condition দিতে code

condition —

true হলে sequentially

false হলে jump

L1:

Sequentially যেতে থাকবো। let 3 instructions পরে branch taken/not taken জানতে পারবো।

condition true হলে W

এরপর normal execution.

3টা instruction time save হলো।

false হলে ঐ 3টা instruction nop করে দিবো।

9 out of 10 times win.

Stall on Branch (Improved)

- Wait until branch outcome determined before fetching next instruction
- 17% branch
- How much is the slowdown?

83% => 1 cycle

17% => 2 cycles

So, effective CPI

$$= \frac{1 \times 83 + 2 \times 17}{100}$$

$$= \frac{83 + 34}{100}$$

$$= 1.17$$

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

Prediction incorrect

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

Delayed Branch

- Always execute the next sequemtial instruction
  - i.e., taken branch is always delayed
  - The delay is hidden by rearrangement

MIPS Solution to Control Hazard

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

MIPS Pipelined Datapath

uneven slicing:

200 100 200 200 100

যত লম্বা হবে max cycle length তত হওয়াই লাগবে।

IW এর length

Total = 800 ps

5টা stage

∴ pipeline এ 160ps লাগবে।

(800/5)

stage বাড়লে time কমবে। pipeline লম্বা হবে বাড়াই না কেন?

MEM

WB

Right-to-left flow leads to hazards

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

Pipeline registers

We may need to update the sizes as we proceed!!!

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

What is the size of the pipeline registers?

IF for Load

lw
Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

ID for Load

আগের pipeline থেকে read
পরেরটায় write

left G shadow — write
right G shadow — read

EX for Load

lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

MEM for Load

single cycle ৪ 800 ps লাগে
— load এর জন্য full cycle ই লাগে

IM থেকে data নিয়ে ভাগ ভাগ করে জায়গা মতো পৌঁছানো
— decode

WB for Load

lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

EX for Store

sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

MEM for Store

WB for Store

SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

WB for Load Revisited

Wrong register number

write register – ID থেকে generate হয়।
WB stage এ ID change হয়ে যায়, so, এটাকেও save করে propagate করে নেওয়া লাগবে।

Corrected Datapath for Load

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Multi-Cycle Pipeline Diagram

- Form showing resource usage

যেকোনো stage এর i/p → previous stage এর o/p

but previous stage এর current o/p আমার input না।

আগের stage এর o/p গুলো somehow store করা লাগবে।

↳ pipeline register

↓

আগের stage এর o/p store

next stage এ supply করবে।

প্রতিটা stage এর মাঝখানে register in করলাম।

— clock আসলে resource update হবে আর আগের value টা register এ stored হবে।

Multi-Cycle Pipeline Diagram

- Traditional form

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

mem আর WB stage এর মাঝে আছে তাই এমন নাম। register এর।

Q. PC + 4 কেন সামনে propagate করছি?

— brach calculate করার জন্য PC+4 কে লাগবে। 3 stage সামনে পর্যন্ত লাগবে।

তার মানে মাঝে আরো 3টা instruction ঢুকবে।

mem stage এ থাকাকালীন IF হবে PC + 4 + 12

= PC + 16

beq করার সময় PC+4

but এখানে PC+4 লাগবে।

PC+16 হয়ে যাচ্ছে IM

Q. → PC−12 করলেই হচ্ছে। কিন্তু এটায় সমস্যা কি?

— extra ALU লাগবে IF এ। সেটাও যদি দেওয়া হয় তবুও কাজেলা। যদি এই 3 instruction এর কোনোটা যদি branch হয় তাহলে সেটা PC +12 আর থাকবে না। তাই PC+4 carry forward করেই যাওয়া লাগবে।

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

Pipelined Control (Simplified)

control ছিল = 10টি
jump বাদে 9টি

ID: No Special Control Needed

— reg write WB stage এর কাজ
so এখানে লাগবে না।

MEM Stage

WB Stage

Confusing!!!
RegWrite is actually at WB stage

IF: No Special Control Needed

EXE Stage

control বের করতে opcode লাগে।

তাই IF এ control generate করা যাবে না।

EX এ করলে too late। এখানেই 4টা control লাগে।

তাই ID তে control generate করা লাগবে।

↳ এখানে control লাগে না

→ এখানে op code ও পেয়ে যাচ্ছি।

→ generate করার পর সেটা register এ store করা লাগবে।

কারণ ID আসলেই নতুন control generate হতে থাকবে।

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

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

Dependencies & Forwarding

$2 has 10
$1 has 30
$3 has 50

A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs

reg 2 clock cycle 5
ready 1
লাগবে 6 এ।

sub $2, $1, $3

and $12, $2, $5 → এই stage এ যে hazard টা হচ্ছে, forwarding লাগবে সেটা বুঝবো কিভাবে?

— আগের instruction এর rd পরের instruction এর rs/rt হলে hazard.

hazard হবে।

$$\begin{cases} EX/MEM.rd = ID/EX.rt \\ \quad \text{or} \quad ID/EX.rs \\ MEM/WB.rd = ID/EX.rs \\ \quad \text{or} \quad ID/EX.rt \end{cases}$$

Forwarding detection:

write নাই।

register এ আছে 5

ALU calculate করে 3

forward করছি 3। conflict!!!

check-1

এই stage এর reg write on থাকলে forward করবো

নাহলে করবো না।

check-2 destination register যদি zero register হয় তাহলেও forward করবো না।

ALU o/p = 3

zero register এর value = 0

forwarding থাকলে conflict হতো।

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

Dependencies & Forwarding

$1 has 30
$3 has 50

A register value written at the first clock cycle, when such a write occurs

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

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

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

forwarding — correct data choose করা লাগবে।

Rs এর জন্য 3 টা option → 1. normal flow থেকে (pipelined register)
2. 2 stage আগে থেকে (Mem/WB)
3. Ex/Mem থেকে

Rt এর জন্যেও 3 টা option

a. No forwarding

Forwarding Paths

rs

forwarding না হ

pipelined

00 → prev reg থেকে

01 → mem/wb

10 → ex/mem

prev থেকে value

non-forwarding এ

forwardA = 0

We have not shown regWrite and $zero checks here.

b. With forwarding

EX/MEM.RegisterRd = ID/EX.RegisterRs

Forwarding Paths

We have not shown regWrite and $zero checks here.

b. With forwarding

MEM/WB.RegisterRd = ID/EX.RegisterRt

sub

ID/Ex এর rs = Ex/Mem এর rd

(Hazard detected)

But forwarding

করবো কোন

mux দিয়ে?

- forward A 10 (Ex/Mem থেকে) → rs

forward B 00 (pipelined value) → rt

<u>sub</u>

forward A — 00

forward B — 01 (rt তে hazard ছিল)

Forwarding Conditions

- EX hazard
  - if
  - M
  - if

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Double Data Hazard

একই register এ write করছি
সেটা আবার read করতে চাই

- 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

Double Data Hazard

CC:		cc1	cc2	cc3	cc4	cc5	cc6	cc7
$2:		30	30	30	30	30	30	30
$3:	20	20	20	20	20	20	20	
$4:	4	4	4	4	4	4	4	
$1:		10	10	10	10	10/40	40/60	60/64

What is expected?

add $1,$1,$2

dest

add $1,$1,$3

dest

add $1,$1,$4

src

RAW hazard

60 is Forwarded Earlier value; WRONG!!!

64 is Forwarded Correct!!!

এখানে forwarding লাগবে।

1, 2 থেকে forwarding হয়েছে। value থাকে।

ID/Ex এর rs = Ex/Mem এর rd
ID/Ex এর rs = Mem/WB এর rd

} ২টা condition satisfy করলো
↓
Hazard condition

→ latest টা আসা উচিত।

Mem/WB তে clock cycle এর old value আছে।

Ex/Mem এ latest value আছে। Mem/WB এর forwarding আটকাবো how?

— check করবো Ex/Mem এর condition already satisfied হয় তাহলে আর forwarding হবে না।
(not and)

recent data তে hazard হলে old value আর forward করবো না।

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 -> Store (Mem to Mem Copy)

rt rs

lw $2, 20($1)

immediate next instruction

rs rt

sw $2, 15($3)

When is data ready?

– DM এ (4th cycle এ)

→ actually data write এখানে হবে।

When is data needed?

– একই cycle এ পারবো না।

এখান থেকে forwarding হবে।

R type

sub $2, $1, $3

and $12, $2, $5

Assignment:

Detection condition + forwarding কিভাবে হবে?

= (IF/ID.MemWrite) and (ID/EX . MemRead) and (IF/ID . registerRt = ID/EX . registerRt)

Load -> Store (Mem to Mem Copy)

Load-Use Data Hazard

Soln { (i) reschedule
(ii) One instruction stall then forwarding (✓)

only forwarding দিয়ে হয়না। future এ load এমন value forward করতে পারিনা।

Need to stall for one cycle

Ex এ check না করে ID তে কেন নিলাম?
→ stall লাগবে 1 এ। তাই Ex এরও আগে check দিতে হবে।

Detection:

Mem/WB এর ~~rd~~ rt = ID/Ex এর rs/rt

① load instruction কিনা check করা লাগবে আগে —

mem read asserted কিনা দেখলেই হবে।

Forwarding unit:

২টা mux দিয়ে

- ID stage এ control signal generate করি।
- stall দেয়া মানে control value 0 করা লাগবে কোনো কিছু write হবে না। mem/reg এ কোনো state change হবে না

1. সব control 0
2. PC write বাদ
3. reg (IF/ID) write বাদ

1 এ stall দেয়া মানে → 2 এর PC আর 3 reg এর value সামনে নেওয়া লাগবে কারন এ দুটো ছাড়া বাকি সব unchanged.

– Hazard detection unit লাগবে

Load-Use Data Hazard

Load-Use Data Hazard

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

Load-Use Data Hazard

Need to stall for one cycle

How to stall???

How to Stall

ID/EX.MemRead is asserted
ID/EX.rt = IF/ID.rs = $2

Make all control fields 0.

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 lw
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

Stall/Bubble in the Pipeline

AND and OR instructions repeat in clock cycle 4 what they did in clock cycle 3:

- AND reads registers and decodes, and
- OR is refetched from instruction memory.

Or, more accurately…

Datapath with Hazard Detection

FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

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

→ predictor লাগবে। not taken — sequential load

branch হবেই না

Branch Hazards

— 3 stage আগে চলে যায়, mem এ detect করি।

3 stall + flush — কারণ reg এ already garbage চলে গেছে

- If branch outcome determined in MEM

Branch Scenario (RECALLING)

Reducing Branch Delay

→ 3 stage stalling এরজন্য
— সবচেয়ে fast
detect করতে পারাবো
হতে।

- 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

Puts the bubble by zeroing the instruction field

Clock 3

Example: Branch Taken

beq $1, $3

XOR করে

32bit OR করবো

— find OR 0 → branch

before<2> taken

1 → not taken

প্রথম 1 টা স্কিপ হবে if branch taken.

66% improve

Puts the bubble by zeroing the instruction field

Clock 3

যদি আগের instruction এর rd $1 হয় সেই $1 পাবো 5th stage এ গিয়ে।

— আরো hazard নিয়ে আসে।

ID তে বুঝবো branch taken কি না

ততক্ষনে IF এ new instruction চলে আসবে।

— stall লাগবে। (pause-resume system)

↳ সামনের সব control 0 (write হবে না)

PC এর write বাঁধা দিচ্ছি

But branch এর ক্ষেত্রে এভাবে হবে না।

bec. pause করে resume করে same জায়গা থেকে।

→ beq কে সামনে যেতে দিতে হবে। stall এর মত আটকে লাভ নেই।

Soln: control কে 0 করবো না।

IF/ID তে 0 ঢুকিয়ে দিবো। flush

flush → branch instruction and brach taken হলে একটা nop ঢুকাতে হবে। যে instruction টা ভুল ছিল সেটা flush করে 0 ঢুকাবো।

Example: Branch Taken

Data Hazards for Branches

immediate পর

soln: (stalling + forwarding)

Stall দিবো, pause হবে, এক clock cycle পর এখান থেকেই resume হবে

forwarding path: এখানে mux দিবো forwarded value নাকি reg value যাবে সেটার জন্য,

Detection condition কি হবে?
forwarding path কেমন হবে?

Data Hazards for Branches

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

add $1, $2, $3

add $4, $5, $6

… <non-dependent>

beq $1, $4, target (immediate পরের instruction না হলে stalling লাগে না, forwarding দিয়েই done)

- Can resolve using forwarding

add $1, $2, $3

add ~~$4~~ $1, $5, $6

… <non-dependent>

beq $1, ~~$4~~ $1, target

— double data hazard হবে

২ জায়গা থেকে forwarding হয়ে different value আসবে।

Soln আগের মতই।

Data Hazards for Branches

load → MEM ৪ ready হয়
beq → ID তে
2 cycle stall লাগবে।
then forwarding.

IF.Flush

All controls made zero

When is data ready?

When is data needed?

lw $1, addr

beq $1, $4, target

beq stalled

lw
instruction
beq.

৪বার stall

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
— instruction
branch } 1번 stall

lw
>2 instruction { — instruction
— 아무거나 instruction
branch } no stall

lw
branch } 2번 stall

R
branch } 1번 stall

R
instruction
branch } 0 stall

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

ID তে এসে branch predict করে লাভ নেই। কারন এখানেই branch taken/not taken calculate হয়। IF এই predict করা লাগবে। But সেখানে instruction decode ই হয় নাই।

Soln: without decoding branch কি না বোঝা লাগবে। branch কি না জানি না।

IF এই: PC এর lower 12 bit save

bht → branch history table এ দেখবো

এই 12 bit আছে কি না। থাকলে দেখবো branch predictor 0 নাকি 1।

not taken ← 0 , 1 → taken

bht fill করতে ID তে যাওয়াই লাগবে।

current PC → beq এর পরের ins.

−4 করে beq এর PC

যত কম bit রাখবো — conflict হবে বেশি

বেশি রাখলে — comparison এ বেশি time লাগবে।

} trade off করা লাগবে then bht এ set করা লাগবে

top tested loop

not taken first 9 times — predictor 0 দিবে

10 th ও → 0 দিবে (আগের ti)

but taken actually

(wrong prediction)

1 করে দিবে (পরের বার 1 থাকে)

so পরের বার ঢোকার সময় 1 দিবে

but not taken

(আবার wrong predictor)

1 bit predictor — ঢোকার সময় একবার

বের হওয়ার সময় একবার

ভুল prediction করবে।

1-Bit Predictor: Shortcoming

- Consider a loop branch:
  - branches nine times in a row
  - then is not taken once.

90% time Taken

- What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

80% time Accurate

2-Bit Predictor

→ error টাকে 1 এ আনে। (1 bit এ 2 ছিল)
2 বার ভুল করলে change করে।

- Only change prediction on two successive mispredictions

a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once.

Branch Delay Slot

প্রতিটা beq instruction এর পরে থাকে, এমন instruction থাকবে যেটা execute করলে সমস্যা হবে না।

Best Option

For (b) & (c):

- usually the target instruction will need to be copied because it can be reached by another path.
- it must be OK to execute the sub instruction when the branch goes in the unexpected direction.
- By "OK" we mean that the work is wasted, but the program will still execute correctly

BDS is scheduled with an independent instruction from before the branch.

BDS is scheduled from the target of the branch

BDS may be scheduled from the not-taken fall-through

branch equal এর target এ নাই এমন একটা destination যুক্ত instruction

beq এর উপরে থাকলে সেটাকে নীচে আনবো।

b,c তে → okay / safe instruction → compiler এর কাজ

branch delay slot পূরণ করা। eg. temp reg এ কাজ

↳ code reschedule

(stall এর জায়গায়)

branch এর immediate পরের slot.

Branch predictor → IF এ বলবে branch taken,

target address next instruction

↳ ID তে calculate হয়

but IF তখনো next sequential instruction load. — ঝামেলা!!

1 cycle stall

target address : IF এই জানা লাগবে।

btb → branch target buffer

pc র lower 4 bit

তার target address সেখানে store করবো।

predictor branch taken বললে এটাও IF এ পাঠাবো

IF এ branch instruction and branch taken জানতে পারলে mux দিয়ে →

btb তে target instruction কেই store করবো। IM ও লাগবে না

target address ও রাখবো না। যেহেতু 32bit ই use করছি so address না রেখে instruction কেই রাখবো।

(i) load-use এ 1 stall ছিল।

but load আমরা একবারই করছি

কিন্তু branch multiple time হতে পারে। তাই 1 cycle stall চাই না।

(ii) pipeline যত লম্বা হবে branch এর জন্য stall এর count ও তত বড় হবে।

CPU → memory হিসেবে cache কে use করে। RAM না।

byte level এ

cache এ sequential instruction থাকে।

branch হলে এসব instruction ফেলে RAM থেকে নতুন load। (যদি branch এর জন্য cache এ না থাকলে)

cache flush 100% sure হয়ে করা লাগবে।

↳ cache to memory — hard drive থেকেও আসতে পারে

physical memory

↳ access করা costly operation → avoid করা লাগবে

harddrive থেকে cache এ আনলাম। কিন্তু ভুল branch predict।
আবার flush করে load করা লাগবে।

primary memory — volatile memory

— OS থাকে না এখানে।

— harddrive এ থাকে। কোন part এ থাকবে Bios এ বলা থাকে।

SDD — logical memory

branch instruction — memory fetching এর সাথে related

costly operation

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

More on Predicting

local → প্রতিটা branch এর জন্য আলাদা predictor.
global

- **correlating predictor:** A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

More on Predicting

- **Tournament predictor:** A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Exception, Interruption

→ external কোনো reason এ execution break করে।

internal/external কোনো reason এ execution break করে।

exception:

eg. · int add এ overflow → আর resume করা যাবে না। kill করা লাগবে।

· user থেকে input নেওয়ার জন্য execution থামিয়ে রাখতে হবে।

→ resume করতে হবে। terminate করা যাবে না।

OS routine লিখে।

mechanism-1 entry point single (যেকোনো interrupt/exeception এর)।

OS এরপর type দেখে routine perform করবে।

cause of interruption বলতে হবে

mechanism-2 একাধিক entry point

যে type এর problem সেটার handler routine এ jump.

— vector interrupt (intel family processor রা use করে)

- MIPS এ single entry point use করে।
- resume করবো যেখানে exception হবে সেই instruction থেকে

epc → exception program counter

↳ instruction এর address টা রাখবে।

যেখানে exception হয়।

(i) Undefined opcode → ID stage এ বুঝবো

(ii) Integer overflow

(iii) memory address invalid → mem stage এ

PC+4

branch/jump

single entry point

exception হলে সামনের সব control 0 করে PC তে single entry point load.

পিছনের যেসব wrong instruction ঢোকে সেগুলো flush করা লাগবে।

(i) সব control 0

(ii) পিছনের সব pipeline register 0 / flush করবো

(iii) PC তে single entry point address

exception handling in these three steps.

exception handler pause register এর value দেখে বুঝবে resumable কি না।

resume করা গেলে Epc তে jump.

PC −4 করে.

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  add $1, $2, $1

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Datapath with controls to handle exceptions

Exception Example

- Exception on add in

```
40    sub  $11, $2, $4
44    and  $12, $2, $5
48    or   $13, $2, $6
4C    add  $1,  $2, $1
50    slt  $15, $6, $7
54    lw   $16, 50($7)
…
```

- Handler

```
80000180    sw   $25, 1000($0)
80000184    sw   $26, 1004($0)
…
```

Exception Example

Exception Example

একই সাথে multiple exception

(i) integer overflow → EX ↳

(ii) invalid opcode → ID ↳

(iii) cache miss error → IF

address করবে earliest exception first.

EX

↓

ID

↓

Handle হলে next exception ↳ যাবে,

না হলে terminate হবে।

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

correct value EPC তে store না করে current value রাখা হয় —imprecise exception

EPC: Precise vs. Imprecise

hardware এর জন্য difficult → correct value রাখা
current value রাখবে।

Exception handler current value আর
cause reg এর value দেখে
actual epc বের করবে।

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue per clock cycle 4টি ins.
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

$\frac{4 \times 10^9}{16 \times 10^9}$

CPI > 1 হয় always

তাই IPC আনা হয় IPC → instruction per clock

→ at least 2 হতে হয়। (moderate version)

Multiple Issue

- Static multiple issue → reschedule/ordering এমন ভাবেই করবে যেন dependency না থাকে
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots" → (compile time এ hazards solve)
  - Compiler detects and avoids hazards
- Dynamic multiple issue → Queue থেকে processor choose করবে (run time এ) কোন কোন independent ins. execute করবে।
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

বিভিন্ন pattern observe করে /intuition থেকে যারা seemingly independent কিন্তু আসলে dependent তাদের আলাদা execution এ দিবে। (independent ভেবে) যদি dependent হয় তাহলে rollback হবে। নাহলে done. speculate করে করবো ভুল হলে rollback

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load followed by a store (whether the load is dependent on the store)
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for ignoring exceptions until it is clear that they really should occur
- Dynamic speculation
  - Can buffer exceptions until instruction completion
    - i.e., the instruction is not speculative anymore

Static Multiple Issue

→ dual issue

2টা ins. pair করে push করবো { → R type / branch, → lw/sw

→ hardward support নেই (২টা ins. একসাথে দেয়ার)

- Compiler groups instructions into "issue packets"
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove/reduce/prevent some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - This will be handled by Hardware (forwarding/Stall)
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue

memory কে double করি না একই সাথে 2 ins. memory execute করতে দিবো না ওভাবেই pair করা।

এখানে stall দেয়া মানে
2টা instruction ই থেমে যাবে।
penalty টা high.

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - add $t0, $s0, $s1
      load $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - (between two issue packet)
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)        # $t0=array element
      addu $t0, $t0, $s2      # add scalar in $s2
      sw   $t0, 0($s1)        # store result
      addi $s1, $s1,–4        # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

load-use hazard
তাই lw এর পরে addu
রাখলাম না।

independent operation.

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,–4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

dual issue

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

একটা loop at least n times চলবে (n=4)
. loop body র n টা copy বানাবো।
register rename করা লাগবে নাহলে conflict হবে।

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, –16 4 টা copy করছি	lw $t0, 0($s1)	1
	nop addu দিলে hazard	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1, –4
      bne  $s1, $zero, Loop
```

4 বার পর পর একবার jump check করবো। কারণ আমরা জানি minm n=4 times ঘুরবে।
run time এ determine হয়

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help

Simple superscalar vs. VLIW

- The code is guaranteed by the hardware to execute correctly.
  - independent of the issue rate
  - or pipeline structure of the processor.
- In VLIW, recompilation may be required for different processor models;
- In other static issue processors, code should run correctly across different implementations
  - but often so poorly as to make compilation effectively required.

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

→ fetching হবে inorder

→ then functional unit এ চলে যাবে।

→ operand অন্য কোনো functional unit এ নেই। তার মানে independent।
register থেকে load করে use.

→ সব operand ready থাকলে সেই ins. execute.

→ যদি operand dependent হয় তাহলে যার উপর dependent সেটা উপরে পাঠাবে। এই ins.

execute হবে out of order এ
write হবে আবার inorder এ

Dynamically Scheduled CPU

Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
      - The name of the functional unit is tracked
    - Register update may not be required

Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - support speculation on load addresses,
  - allowing load-store reordering,
  - Don’t commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
- Some parallelism is hard to expose
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

ARM Cortex-A8 Pipeline

→ 12단 stage pipelining
G.T

Address Generation Unit

Branch Target Buffer
Global History Buffer
Return Stack

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall