CS2100

Computer Organization

AY2022/23 Semester 1

Notes by Jonathan Tay

Last updated on October 31, 2022

Contents

Part I

Data Representation

Data is represented as a sequence of **bits** — 0 or 1.

8 bits form a **byte**, and a fixed multiple of bytes in a computer architecture forms a **word**.

number of bits $\Longleftrightarrow$ number of values

n bits can represent up to 2^n values.

Conversely, we need at least $\lceil \log_2 m \rceil$ bits to represent m values.

1 Number Systems

Number systems are **weighted-positional** systems, with the **decimal** number system having a **base** (or **radix**) of 10.

number system	base	prefix
binary	2	0b
octal	8	0
decimal	10	—
hexadecimal	16	0x

The digit to the left of the decimal point has a weight of 10^0, with the exponent *incrementing* leftward and *decrementing* rightward.

decimal $\Longrightarrow$ binary

For *whole numbers*, use **repeated division-by-two**:

1. divide the value by 2, and record the remainder
2. prepend the remainder to the binary representation
3. repeat until the value is 0

For *fractions*, use **repeated multiplication-by-two**:

1. multiply the value by 2, and record the carry
2. append the carry to the binary representation
3. repeat until the value is 1
4. append the 1 to the binary representation

binary $\Longrightarrow$ decimal

Use **weighted summation**:

1. assign the digit to the left of the binary point a weight of 2^0
2. increment the weight exponents leftward
3. decrement the weight exponents rightward
4. multiply each digit by its weight
5. sum the products

These conversions generalize to any base!

Converting between 2 bases can be done with base-10 as an intermediary.

binary $\Longleftrightarrow$ base-2^k

1. partition the binary representation into k-bit chunks outward from the binary point
2. convert each chunk to base-10
3. convert each base-10 value to base-2^k

Apply the reverse direction for converting from base-2^k to binary.

1.1 Negative Numbers

Signed numbers include all positive and negative values, unlike **unsigned numbers** which only include non-negative values.

We will consider n-bit values in this section.

sign-and-magnitude

The MSB is the **sign bit** — 0 for positive, 1 for negative.

Negate a value by inverting the sign bit.

- **range**: $-2^{n-1} + 1$ to $2^{n-1} - 1$
- **zeros**: -0_{10} and $+0_{10}$

1s complement

An n-bit number x has a negated value $-x = 2^n - x - 1$.

Negate a value by inverting *all the bits*.

- **range**: $-2^{n-1} + 1$ to $2^{n-1} - 1$
- **zeros**: -0_{10} (all 1s) and $+0_{10}$ (all 0s)

The weight of the MSB is $-2^{n-1} + 1$.

2s complement

An n-bit number x has a negated value $-x = 2^n - x$.

Negate a value by inverting all the bits to the left of the *rightmost* 1.

- **range**: -2^{n-1} to $2^{n-1} - 1$
- **zeros**: 0_{10} (all 0s)

Unlike sign-and-magnitude and 1s complement, 2s complement does not have duplicate zeros.

The weight of the MSB is -2^{n-1}.

The MSB also represents the sign — 0 for positive, 1 for negative.

Negating a fractional number in a complement representation is no different than negating a whole number!

excess representation

Distribute positive and negative values by a simple addition or subtraction by k, which is also known as the **bias**.

k is typically $2^n - 1$ for an n-bit number.

The binary value in excess-k *subtracts* k from its decimal value.

Conversely, the decimal value in excess-k *adds* k to its binary value.

1.2 Overflow

Overflow occurs when addition or subtraction goes beyond the fixed range of a signed number.

Subtraction is equivalent to addition of the negated value.

Note that in 1s addition, the carry out of the MSB is *added* to the result, but this carry out is *discarded* in 2s addition.

detecting overflow

If the MSB (sign bit) of the result is different from the MSBs of the operands, then overflow has occurred.

Note that overflow can only occur when the operands have the same sign, i.e., adding a negative number to a positive number will *never* result in overflow.

1.3 Real Numbers

Fixed-point representation fixes the number of bits for the whole number and fractional parts, which limits the range of values.

The IEEE 754 **floating-point representation** resolves this by allocating a fixed number of bits to the **sign**, **exponent**, and **mantissa**.

precision	bits	sign	exp.	mantissa	bias
single	32	1	8	23	127
double	64	1	11	52	1023

The **mantissa** is **normalized** with an implicit leading 1 bit, e.g. 110.1_2 is normalized to $1.101_2 \times 2^2$, and only 101 is stored in the mantissa.

decimal $\implies$ IEEE 754 floating-point representation

1. convert the decimal value to normalized binary, i.e., $\pm 1.xxx \times 2^y$
2. determine the sign bit — 0 for positive, and 1 for negative
3. add the bias to the exponent y — $+127$ for single precision, and $+1023$ for double precision
4. convert the exponent to binary
5. concatenate the sign bit, exponent, and mantissa xxx

Part II

Instruction Set Architecture

2 ISA Overview

An **instruction set architecture** (ISA) is an abstraction on the interface between hardware and low-level software, which runs on a **processor**.

Processors are connected to **memory** (RAM) via a **bus**, across which code and data is transferred.

ISAs have two major design philosophies:

1. **CISC** — Complex Instruction Set Computer:
   – single instructions for complex operations
   – smaller program sizes
   – complex implementation, little room for hardware optimization
2. **RISC** — Reduced Instruction Set Computer:
   – smaller and simpler instruction set
   – software combines simpler operations to implement high-level language statements
   – room for compiler optimization

2.1 Data Storage Architecture

1. **stack architecture**:
   operands are implcitly popped from the stack
2. **accumulator architecture**:
   one operand is implcitly stored in an accumulator
3. **general-purpose register architecture**:
   operands are stored explicitly in registers, operations are register-memory or register-register
4. **memory-memory architecture**:
   operands are read from memory

2.2 Memory Architecture

Memory transfer takes place across buses:

1. **address bus**: k-bits, uni-directional from processor to memory
2. **data bus**: n-bits, bi-directional
3. **control lines**: bi-directional, e.g. read/write controls

The address bus feeds addresses from the **memory address register** to the memory.

Data is written to or read from the **memory data register**, depending on the R/W control line.

Endianness is the relative ordering of the *bytes* in a word (*not* the bits in a byte!) in memory:

– **big-endian**: MSB stored in smallest address
– **little-endian**: LSB stored in smallest address

3 Instruction Set Encoding

Instructions consist of an **opcode** — a unique code to identify the operation — as well as **operands**.

In an ISA with **fixed-length instructions**, we need to fit multiple sets of instruction types each with the same number of bits.

We use the **expanding opcode** scheme in which the opcode has variable length for different instructions.

maximizing the total number of instructions

Maximize the number of instructions in each set, starting from the *largest* set to the smallest.

minimizing the total number of instructions

Maximize the number of instructions in each set, starting from the *smallest* set to the largest.

Part III

MIPS

4 Registers

Memory access is slow. To avoid frequent memory access, temporary values are stored in the processor in **registers**, which are limited in number.

Registers *do not have data types*, unlike program variables. Instructions always assume that the data stored in a register is of the correct type.

MIPS has **32 registers**, which are referred to by *number* or *name*:

name	#	usage
$zero	0	constant value zero
$at	1	reserved for the assembler
$v0 - $v1	2 - 3	values for results and expression evaluation
$a0 - $a3	4 - 7	arguments
$t0 - $t7	8 - 15	temporaries
$s0 - $s7	16 - 23	program variables
$t8 - $t9	24 - 25	temporaries
$k0 - $k1	26 - 27	reserved for the OS
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

5 MIPS Assembly Language

The **general instruction syntax** is as follows:

op $s0, $s1, $s2	
	description
op	operation
$s0	destination register
$s1	source register 1
$s2	source register 2

Each instruction executes a *single* command. Each line of assembly code contains *at most* one instruction.

Almost all MIPS operations are *register-to-register*

The # hex synbol is used for comments.

5.1 Arithmetic Instructions

instruction	format	opcode/funct
add $rd, $rs, $rt	R	$0/20_{hex}$
sub $rd, $rs, $rt	R	$0/22_{hex}$
addi $rt, $rs, imm	I	8_{hex}
move $s0, $s1	psuedo	—

immediate subtraction

There is no subi operation as it is equivalent to addi with a negative constant.

assigning variables

The move instruction is a **psuedo-instruction** which is translated into its MIPS equivalent by the compiler:

add $s0, $s1, $zero

To assign a constant imm to a register:

addi $s0, $zero, imm

The immediate values in **immediate operations** such as addi range from -2^{15} to $2^{15} - 1$, as the 16-bit 2s complement system is used.

setting large constants

If 32-bit constants are required, use the lui operation to load the most-significant 16-bits first, followed by an ori operation to set the least-significant 16-bits:

```
lui $t0, 0xAAAA       # t0 = 0xAAAA0000
ori $t0, $t0, 0xF0F0  # t0 = 0xAAAAF0F0
```

Use the **temporary registers** $t0 to $t9 to store intermediate results in complex expressions.

5.2   Logical Instructions

instruction	format	opcode/funct
sll $rd, $rt, shamt	R	$0/00_{hex}$
srl $rd, $rt, shamt	R	$0/02_{hex}$
and $rd, $rs, $rt	R	$0/24_{hex}$
or $rd, $rs, $rt	R	$0/25_{hex}$
xor $rd, $rs, $rt	R	$0/26_{hex}$
nor $rd, $rs, $rt	R	$0/27_{hex}$
andi $rt, $rs, imm	I	C_{hex}
ori $rt, $rs, imm	I	D_{hex}
xori $rt, $rs, imm	I	E_{hex}
lui $rt, imm	I	F_{hex}

Bitwise NOR sets the result to 1 if both bits are 0, and 0 otherwise.

Bitwise XOR sets the result to 1 if both bits are different, and 0 otherwise.

> **bitwise NOT**
>
> There is no not operation as it is equivalent to either of the following:
>
> – nor $rd, $rs, $zero
> – xor $rd, $rs, $rt, where $rt is all 1s

The absence of nori keeps the instruction set small.

In bitshift operations (sll and srl), the empty positions are filled with zeros, and the **shift amount** is limited to **5 bits**.

> **multiplication and division**
>
> For multiplication/division by k, if k is a power of 2, use sll and srl respectively, setting the shamt to k.
>
> Otherwise, use a loop.

5.3   Memory Instructions

Memory can be thought of as a *single-dimensional array* of memory location, with each having an **address**.

Memory addresses allow access to *bytes* of data, or **words** of data, which are usually 2^n bytes — the common unit of transfer between the processor and memory.

Word alignment occurs in memory when words begin at a *byte address* which is a multiple of the word size — 2^n bytes.

In MIPS, each word is 32 bits (4 bytes), and addresses are 32-bits long — such that 2^{30} words are addressable, each of which differing by 4.

instruction	format	opcode/funct
lw $rt, offset($rs)	I	23_{hex}
sw $rt, offset($rs)	I	$2B_{hex}$
lb $rt, offset($rs)	I	20_{hex}
sb $rt, offset($rs)	I	28_{hex}
ulw $rt, offset($rs)	pseudo	—
usw $rt, offset($rs)	pseudo	—

For the word-aligned memory instructions lw and sw, the resulting address of $rs + offset must be a multiple of 4.

The offset is a 16-bit 2s complement number.

5.4   Control Flow Instructions

instruction	format	opcode/funct
beq $rs, $rt, label	I	4_{hex}
bne $rs, $rt, label	I	5_{hex}
j label	J	2_{hex}
slt $rd, $rs, $rt	R	$0/2A_{hex}$
slti $rt, $rs, imm	I	A_{hex}
blt $rs, $rt, label	pseudo	—
bgt $rs, $rt, label	pseudo	—
ble $rs, $rt, label	pseudo	—
bge $rs, $rt, label	pseudo	—

slt and slti sets the result to 1 if $rs < $rt or imm, and 0 otherwise.

beq and bne are **conditional jumps** — they jump to the label if the condition is true.

Labels are written as <label>: to the left of a statement, but they are *not* instructions.

A j instruction is equivalent to beq $s0 $s0, <label>.

The **program counter** (PC) typically stores the address of the *next* address to be executed, and has to be modified by the branching and jump instructions.

6   Instruction Encoding

Every MIPS instruction is **32 bits** in 3 possible formats:

format	source registers	destination registers	immediate values
R	2	1	0
I	1	1	1
J	0	0	1

6.1   R-format

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

– `opcode := 0` for all R-format instructions,
– `funct` determines the instruction,
– `shamt := 0, rd := arith(rs, rt)` for non-shift (arithmetic) instructions, and,
– `rs := 0, rd := shift(rt, shamt)` for shift instructions.

6.2 I-format

	opcode	rs	rt	immediate
bits	6	5	5	16

– `opcode` determines the instruction since there is no `funct` field,
– `rt` determines the *destination* register since there is no `rd` field,
– `immediate` is a *signed* integer in 2s complement except for bitwise operations where it is *unsigned*,
– `rt := op(rs, immediate)` in general for all non-branching instructions, and,
– `PC := (PC + 4) + (immediate * 4)` when branching, otherwise `PC += 4`, which is the next instruction.

For branching instructions, `immediate` is the offset from the *next* instruction to the label of the *target* instruction.

PC is incremented in multiples of 4 due to word-alignment, which also means that we can now branch $2^{15} \times 4 = 2^{17}$ bytes away from PC.

6.3 J-format

	opcode	target address
bits	6	26

The last 2 bits of every instruction are always `00` due to word-alignment, so we leave them out of the target address.

This leaves with an effective range of 28 bits for the target address, and the remaining 4 bits are derived from the most significant (leftmost) bits of `PC + 4`.

The **destination address** is therefore:

	(PC+4)[0:4]	target address	00
bits	4	26	2

This creates a maximum jump range of 256 MB.

6.4 Addressing Modes

Addressing modes are used to calculate the address of an operand.

1. **register addressing** — `add, xor, etc.`:
   operand is a register
2. **immediate addresssing** — `addi, andi, etc.`:
   operand is a constant within the instruction
3. **base/displacement addressing** — `lw, sw`:
   operand is a memory location at the address the sum of a register and a constant in the instruction
4. **PC-relative addressing** — `beq, bne`:
   address is the sum of the PC and a constant in the instruction
5. **psuedo-direct addressing** — `j`:
   part of the instruction concantenated with part of the PC

7 Datapath

A collection of components that process data, and perform arithmetic, logical, and memory operations.

The **instruction execution cycle** has 5 stages:

1. **fetch**:
   get instruction from memory, address in PC
2. **decode and operand fetch**:
   determine the operation and get the operands needed
3. **execute (ALU)**:
   perform the computations to get a result
4. **execute (memory access)**:
   read from or write to memory if necessary
5. **write-back**:
   store the result of the operation

7.1 Fetch

There are 3 steps in the fetch stage:

1. Use PC to fetch the instruction from memory.
2. Increment PC by 4 to get the address of the next instruction.
3. Feed the output instruction to the decode stage.

The **instruction memory** element is a **sequential circuit** with an internal state which stores the instructions.

When supplied an instruction address m, it outputs the content at address m.

The **adder** element takes in the 32-bit PC and the 32-bit constant 4, and outputs the 32-bit sum `PC + 4`.

The **clock signal** is a *square wave* with *rising* and *falling* edges, and a period controlled by the CPU.

PC is read in the first half of the **clock period** and updated at the *next rising clock edge*.

7.2 Decode

There are 3 steps in the decode stage:

1. Read the `opcode` to determine the instruction type and field lengths.
2. Read data from all necessary registers.
3. Feed the operation and operands to the ALU stage.

The **register file** element is a collection of 32 registers which can be read from or written to.

input	bits	output	bits
read register 1	5	read data 1	32
read register 2	5	read data 2	32
write register	5		
write data	32		

The `RegWrite` control signal determines whether the instruction should read from or write to the registers:

	RegWrite	registers per instruction
read	0	≤ 2
write	1	≤ 1

7.2.1 R-format Decoding

The binary content (`Inst`) of R-format instructions map to the inputs as follows:

- rs ≡ Inst[25:21] ↦ rr1
- rt ≡ Inst[20:16] ↦ rr2
- rd ≡ Inst[15:11] ↦ wr

7.2.2 I-format Decoding

For I-format instructions, 2 problems arise:

1. `rt`, not part of `imm`, needs to be fed to `wr`
2. `imm` needs to be fed to the ALU, not `rd2`

A **multiplexer** chooses the correct `Inst` slice to feed to `wr` using the control signal **RegDst**:

	RegDst	input to wr
I-format	0	rt ≡ Inst[20:16]
R-format	1	rd ≡ Inst[15:11]

- rs ≡ Inst[25:21] ↦ rr1
- rt ≡ Inst[20:16] ↦ rr2
- rt ≡ Inst[20:16] ↦ wr

Another multiplexer chooses the correct 32-bit binary data to feed into the ALU:

	ALUSrc	input to ALU opr2
R-format or beq	0	rr2 ≡ *rt ≡ *Inst[20:16]
I-format not beq	1	imm ≡ Inst[15:0], sign extended to 32-bits

7.3 Arithmetic Logic Unit (ALU)

The ALU performs the arithmetic, shifting, logical, memory, and branching operations.

It takes the operation and operands from the decode stage, performs its computation, and feeds the result to the memory stage.

input	bits	output	bits
operand 1	32	is zero?	1
operand 2	32	result	32

The `ALUControl` signal determines the operation to perform:

ALUControl	operation
0000	and
0001	or
0010	add
0110	sub
0111	slt
1100	nor

7.3.1 Branching Instructions

A multiplexer chooses between the next instruction (`PC + 4`) and the branch target (`BT`) using the control signal **PCSrc**:

isZero	PCSrc	next instruction
false ≡ 0	0	PC + 4
true ≡ 1	1	BT ≡ PC + 4 + imm * 4

Because both register contents need to be compared, the `ALUSrc` control value is set to 0 for branching instructions.

BT is computed as follows:

1. left shift by 2 bits the sign-extended `imm` from the decode stage (multiplying the offset by 4)
2. add it to `PC + 4` from the adder in the fetch stage

7.4 Memory

Only `lw`/`lb` and `sw`/`sb` instructions operate in this stage, the rest remain idle.

The **data memory** element exists in RAM:

input	bits	output	bits
address (addr)	32	read data (rdt)	32
write data (wd)	32		

`addr` is the effective memory address computed by the ALU, and `wd` is connected to `rd2` from the register file from the decode stage.

The control signals **MemRead** and **MemWrite** control the memory operations:

MemRead	MemWrite	action
0	1	write wd into addr
1	0	read contents of addr into rdt
0	0	do nothing
1	1	undefined, should not occur

7.4.1 Non-memory Instructions

A multiplexer chooses between the result of the ALU and the read data from memory, using the control signal **MemtoReg**:

	MemToReg	output
non-memory instruction	0	ALU output
memory instruction	1	read data

This output is fed to the write-back stage.

7.5 Write-back

Stores, branches, and jumps remain idle in this stage as there is nothing to be written.

The result of the memory stage is fed into write data (wd) of the register file.

8 Control

Control signals are generated by a **control unit** based on the type of instruction.

The control unit is a **combinational circuit** which takes in the 6-bit opcode and the 5-bit function code, and outputs the 8 control signals.

funct is only needed to determine ALUSrc; every other control signal can be derived from opcode alone.

8.1 ALUControl

We start with a simplified 1-bit ALU.

1-bit MIPS ALU

4 control signals are needed:

1. **Ainvert**: 1 to invert A, 0 otherwise
2. **Binvert**: 1 to invert B, 0 otherwise
3. **operation (2 bits)**: select one of the 3 results

5 I/O signals:

1. **A**: input
2. **B**: input
3. **Cin**: carry in
4. **Cout**: carry out
5. **result**: output

These 1-bit ALUs are daisy-chained to form a 32-bit ALU.

The carries from one of the 1-bit ALUs is fed out via Cout and into the next one via Cin.

Ainvert and Binvert control the multiplexers which select between their inputs and their negated values:

Xinvert	output
0	X
1	NOT(X)

The operation control signal selects one of the 3 results from the AND gate, OR gate, and the adder ADD:

operation	output
00	AND
01	OR
10	ADD
11	SLT

1-bit subtraction

Cin is set to 1 due to 2s complement:

$$A - B \equiv A + (-B) \equiv A + B' + 1$$

ALUControl

In MSB to LSB order, ALUControl comprises of:

1. Ainvert
2. Binvert
3. operation

8.1.1 ALUOp

It is necessary to use the **multi-level decoding approach** to simplify the design process and size of the main controller.

ALUOp is an intermediate 2-bit control signal which is generated from opcode and used with funct to determine ALUControl.

instruction	ALUOp	funct	ALUControl
lw, sw	00	XXXXXX	0010
beq	01	XXXXXX	0110
add	10	100000	0010
sub	10	100010	0110
and	10	100100	0000
or	10	100101	0001
slt	10	101010	0111

In general, ALUOp is 10 for R-format instructions.

9 Pipelining

Pipelining increases workload throughput, but is *rate limited* by the slowest stage in the pipeline, and *stalled* by dependencies between stages.

Each execution stage in the MIPS datapath is a **pipeline stage** which takes 1 clock cycle:

1. **fetch**: requires the instruction memory
2. **decode**: involves the register file
3. **execute**: involves the ALU
4. **memory**: requires data memory
5. **writeback**: involves the register file

Data required for each stage is stored separately in **4 pipeline registers**:

1. **IF/ID**: between `fetch` and `decode`
2. **ID/EX**: between `decode` and `execute`
3. **EX/MEM**: between `execute` and `memory`
4. **MEM/WB**: between `memory` and `writeback`

register	receives	supplies
IF/ID	`Instruction[PC]`, PC + 4	RR1 and RR2, 16-bit `offset`, PC + 4, WR
ID/EX	RD1 and RD2, 32-bit `imm`, PC + 4, WR	RD1 and RD2, 32-bit `imm`, PC + 4, WR
EX/MEM	PC + 4 + 4 ∗ `imm`, ALU result, `isZero?`, RD2, WR	PC + 4 + 4 ∗ `imm`, ALU result, `isZero?`, RD2, WR
MEM/WB	ALU result, memory data read, WR	WR

control signals

All control signals are generated in the `decode` stage and propagated to their respective pipeline stages.

– **execute**: `RegDst, ALUSrc, ALUOp`
– **memory**: `MemRead, MemWrite, Branch`
– **writeback**: `MemToReg, RegWrite`

9.1   Performance

single-cycle processor

Each instruction takes 1 clock cycle to execute.

The **cycle time** CT_{seq} is the longest time taken to execute any single instruction:

$$CT_{seq} = \max\left(\sum_{k=1}^{N} T_k\right)$$

where N is the number of stages and T_k is the time taken to execute an operation in stage k.

The **execution time** ET_{seq} depends on the number of instructions I:

$$ET_{seq} = CT_{seq} \cdot I$$

multi-cycle processor

Each *stage* takes 1 clock cycle to execute.

The **cycle time** CT_{multi} is the longest time taken to execute any single stage.

The **execution time** ET_{multi} depends on the number of instructions I and the average number of instructions per instruction CPI_{avg}:

$$ET_{multi} = CT_{multi} \cdot I \cdot CPI_{avg}$$

pipelined processor

The **cycle time** $CT_{pipelined}$ incurs a pipelining overhead T_d:

$$CT_{pipelined} = \max(T_k) + T_d$$

where T_k is the time taken to execute an operation in stage k.

The **execution time** $ET_{pipelined}$ depends on the number of instructions I and stages N:

$$ET_{pipelined} = CT_{pipelined} \times (I + N - 1)$$

In an **ideal pipeline**:

– every stage takes the same amount of time:
  $$\sum_{k=1}^{N} T_k = N \cdot T_1$$
– there is no pipeline overhead:
  $$T_d = 0$$
– the number of instructions is much larger than the number of stages:
  $$I >> N$$

This results in a **ideal speedup** by N:

$$\frac{ET_{seq}}{ET_{pipelined}} = \frac{I \cdot N \cdot T_1}{(I + N - 1) \cdot T_1} \approx \frac{I \cdot N \cdot T_1}{I \cdot T_1} = N$$

9.2   Hazards

Pipeline hazards are problems that prevent an instruction immediately after a previous instruction.

Structural hazards are caused by simultaneous use of the same register file, memory, or ALU by multiple instructions.

resolving structural hazards

a) stall the execution by one or more cycles.

b) split the memory into `Data` and `Instruction`.

c) write to the register file in the first half of a cycle, and read in the second half — allowed as register access is fast.

Data hazards are caused by **read-after-write (RAW) dependencies** between instructions, where the result of an instruction is read by a subsequent instruction.

This can lead to **stale results** if the result is not written to the register file in time.

resolving RAW data hazards — forwarding

Suppose sequential instructions B, C, and D each have a RAW dependency on instruction A.

With **forwarding**, the result of A is computed in the `execute` stage, and is fed into the `execute` stage of B and C, bypassing the register file.

B receives the result from the pipeline register EX/MEM. C receives the result from MEM/WB.

Because writing and reading to the register file in two different halves of the same clock cycle, instruction D can read the result from the register file.

resolving loading data hazards — stalling

Suppose instruction A reads data from memory, and B and C are dependent on this data.

Because A produces this data in the `memory` stage only after it is needed by the `execute` stage of B, no data can be forwarded yet.

Consequently, B has to **stall** the pipeline until its `execute` stage can receive forwarded data from MEM/WB from A.

If B is the next instruction after A, then the stall is 1 clock cycle long.

Because writing and reading to the register file in two different halves of the same clock cycle, instruction C can read the result from the register file.

Control hazards are caused by a change in program flow.

Branching decisions are made in the `memory` stage, which is too late as the next instructions will be partially executed.

resolving control hazards — stalling

The **branch decision** is known in the `memory` stage, so we can stall for 3 clock cycles and then `fetch` the next instruction.

resolving control hazards — early branch resolution

Moving the branch decision to the `execute` stage allows reduces the stall duration to 1 clock cycle.

If a value in the branch instruction is *computed* in the preceding instruction, the stall duration is only reduced to 2 clock cycles.

If a value in the branch instruction is *loaded* in the preceding instruction, the stall duration remains at 3 clock cycles.

resolving control hazards — branch prediction

Assume all branches are not taken, no stall required. Flush the succeeding instruction from the pipeline if the branch *is* taken.

Flushing wastes 3 clock cycles without early branch resolution. With early branch resolution, it wastes the same number of clock cycles as stalling would.

resolving control hazards — delayed branching

Instructions which precede the branching instruction and are executed regardless of the branch decision are re-ordered to execute *after* the branch instruction.

The MIPS processor has 1 of these **branch-delay slots**.

If no such instruction can be found, the branch instruction is delayed by 1 clock cycle via a nop.

10   Caching

A **cache hit** occurs when requested data is in the cache, and a **cache miss** occurs when requested data is not in the cache.

Cache misses incur a **cache miss penalty** which is the time taken to replace the cache block in addition to the **hit time**.

average access time

average access time = hit rate × hit time + (1 - hit rate) × miss penalty

The **cache block** is the unit of transfer between the memory and cache, and typically ≥ 1 word.

The word addresses of a cache block of size 2^N bytes have the same $(32-N)$ leftmost bits which form the **block number**. The remaining N rightmost bits form the **byte offset**.

block number $[31..N]$	byte offset $[N-1..0]$

10.1   Direct Mapped (DM) Caches

Given a cache with 2^M **cache blocks**, the **index** are the M rightmost bits of the *block number*, and the **tag** are the remaining leftmost bits of the block number.

tag $[31..N+M]$	index $[N+M-1..N]$	offset $[N-1..0]$

Caches incur a **cache overhead** as each cache block stores the tag of the corresponding memory block as well as a **valid bit** indicating whether its data is valid.

Cache hits only occur when `valid[index] == true` and `tag[index] == block_number`.

cache circuitry

For an N-bit offset and M-bit index, the tag is $32 - N + M$ bits.

1. Index into the cache using the cache index to get the cache block.
2. Check if it is a cache hit via an AND gate.
3. If it is a cache hit, multiplex the words from the cache block using the offset.

Recall that MIPS uses a 4-byte word size, so the rightmost 2 bits of the offset is ignored, and only the 2 leftmost bits are used to index into the cache block.

cache misses

– **compulsory misses**:
occurs on the first access to a block, requiring the block to be brought into the cache

– **conflict misses (DM and SA caches only)**:
occurs when several memory blocks are mapped to the same cache block

– **capacity misses (FA caches only)**:
occurs when blocks are discarded as cache capacity is exceeded

write policies

Writing to a cache can cause data inconsistency, as the data is only modified in cache but not in memory.

– **write-through cache**:
write to memory as well as cache, but add a buffer between cache and memory otherwise data writes operate at the speed of the memory

– **write-back cache**:
writes to cache, and writes to memory when the cache block is evicted determined by a **dirty bit** at each cache block

write miss policies

– **write-allocate**:
load the complete block and change the required word, writing to memory based on the write policy

– **write-around**:
ignore the cache and write directly to memory

10.2 Set Associative (SA) Caches

An N-way set associate cache consists of a number of **sets**, such that each set has N cache blocks.

Each memory block maps to a unique cache set, within which they can be placed in *any* of the N cache blocks.

The sets of the cache are indexed via a **set index**, and *all* N cache blocks of a set has to be searched for a cache hit/miss.

tag $[31..N + M]$	set index $[N + M - 1..N]$	offset $[N - 1..0]$

For a cache of size k and block size b, the number of bits needed for the set index is $\log_2(k \div b \div N)$.

A direct-mapped cache of size N has about the same miss rate as a 2-way associative cache of size $\frac{N}{2}$.

10.3 Fully Associative (FA) Caches

A fully associative cache allows any memory block to be placed in any cache block, but this comes at the cost of searching *every* cache block for memory access.

The tags are equivalent to the block numbers, and all tags are compared in parallel.

FA caches do not suffer from conflict misses.

block replacement policies

A full cache block must be replaced when a new memory block is to be inserted.

– **least recently used (LRU)**:
cache hits record the cache block accessed, replace the block that has not been accessed for the longest time due to **temporal locality**

Other policies include **least frequently used (LFU) random replacement**, and **first-in first-out (FIFO)**.