Midterm Cheat Sheet

By Phanuphat Srisukhawasu
CS2100 Taken in AY2024/25 S2
Last updated: March 11, 2025

Number Systems

Compatible Base Checking

Case 1: Checking base x for addition/subtraction operations.

1. Consider the first digit from LSB where adding digits cause overflow, i.e. the sum is lesser than one of the original values.
2. Apply the formula: `(digit1 + digit2) % x = resultDigit` and find x from here.

Case 2: Checking base for other operations. These problems mostly require brute force.

Complement Systems

Given that n/m is the number of integer/fractional digits

- (r-1)'s complement: $X' = r^n - r^{-m} - X$ (ranging from $-(r^{n-1} - 1)$ to $r^{n-1} - 1$).
- r's complement: $X'' = r^n - r^{-m} - X + 1$ (ranging from $-r^{n-1}$ to $r^{n-1} - 1$).

> **Example**
>
> Calculate $0101.11 - 010.0101$ (in binary).
>
> Solution:
>
> - $0101.1100 - 0010.0101 = 0101.1100 + (2^4 - 2^{-4} - 0010.0101)$
> - $= 0101.1100 + (10000 - 0.0001 - 0010.0101) = 0101.1100 + (1111.1111 - 0010.0101)$
> - $= 0101.1100 + 1101.1010 = (1)0011.0110 = 0011.0111$
>
> Calculate 10's complement of -1 in 4 bits representation.
>
> Solution: -1 can be represented in 9's complement as $10^4 - 10^{-0} - 1 = 9998$ and as $9998 + 1 = 9999$ in 10's complement.

> **Warning**
>
> 1. In base-r complements, digit weights are not applicable except when $r = 2$.

2. Always extend the digits in both integer and fractional parts to match the other operands, as shown in the first example.
3. (r–1)'s complement propagates a carry-out to the end, whereas r's complement will just ignore the carry.

IEEE-754 Representation

Example

Find the decimal value of `0xC4007000`.

Solution:

1. Use the calculator to convert from hex to binary (group by 1, 8, 23 bits): `0b1 10001000 00000000111000000000000`.
2. Read the first bit (`0` is positive and `1` is negative).
3. The next 8 bits are the exponent in Excess-127 format. `0b10001000` = 136 (Excess-127) = 136 - 127 = 9 (in decimals).
4. Write the expression as $\pm 1.XX\ldots X \times 2^n$ where X's are the remaining 23 bits = $-1.00000000111 \times 2^9$ which is `-0b1000000001.11` (like scientific notation).
5. Convert the resulting binary bits into decimal using the calculator: -513.75.

Note: Decimal + 127 = Excess-127 and Excess-127 - 127 = Decimal. This formula can be useful when converting the number back (from step 5. to 1.)

Tip

- The range of the Excess-M system is from $-(M-1)$ to M.
- The smallest positive number representable in the IEEE-754 format is given by: $1.00\ldots0 \times 2^{-126}$.
- The most negative number representable in the IEEE-754 format is given by: $-1.11\ldots1 \times 2^{127}$.

C Programming

C always uses pass-by-value, but we can simulate pass-by-reference with pointers.

Warning

1. An array name (`arr`) is a fixed pointer to its first element (`&arr[0]`), meaning you can't reassign it (`arr1 = arr2` is invalid).

2. When passed to functions, an array decays into a pointer to its first element.
3. `struct` objects are always passed by value (copied in full), except when passing the pointers to the object.
4. Arrays of `struct` objects are effectively passed by reference through pointers.
5. To increment a pointer's value, use `(*p)++`. Writing `*p++` increments the pointer itself (by size in bytes of the corresponding data type), not the value it points to.

MIPS Programming

Instruction Encoding & Decoding

- Encoding: Refer to the instruction sheet, write the entire instruction in binary, and then convert to hex using the calculator.
- Decoding: Convert hex to binary using the calculator. Write the encoded instruction. After noting the first 6 bits, read the actual opcode to determine the type of the instruction. The subsequent groupings depend on whether it is R (5/5/5/5/6), I (5/5/16), or J (26) format.

Register

Aside from the constant zeroes (`$zero`), we have `t` for temporaries and `s` for saved temporaries.

`$t0`	`$t1`	`$t2`	`$t3`	`$t4`	`$t5`	`$t6`	`$t7`	`$t8`	`$t9`
01000	01001	01010	01011	01100	01101	01110	01111	11000	11001
`$s0`	`$s1`	`$s2`	`$s3`	`$s4`	`$s5`	`$s6`	`$s7`		
10000	10001	10010	10011	10100	10101	10110	10111		

R-Format

The instruction can be determined using the `funct` field. The shift amount (`shamt`) is only applicable to shift left/right logical instructions.

Mnemonic	`opcode` (6)	`rs` (5)	`rt` (5)	`rd` (5)	`shamt` (5)	`funct` (6)
`add rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100000
`sub rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100010
`sll rd, rt, shamt`	000000	00000	`rt`	`rd`	`shamt`	000000

Mnemonic	`opcode` (6)	`rs` (5)	`rt` (5)	`rd` (5)	`shamt` (5)	`funct` (6)
`srl rd, rt, shamt`	000000	00000	`rt`	`rd`	`shamt`	000010
`and rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100100
`or rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100101
`xor rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100110
`nor rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	100111
`slt rd, rs, rt`	000000	`rs`	`rt`	`rd`	00000	101010

> **Tip**
>
> The opcode of the R format instruction is always `000000`.

I-Format

The immediate is always a 16-bit integer. You need to use load upper immediate (`lui`) with `ori` (for the lower 16 bits) to extend it to 32 bits.

Mnemonic	`opcode` (6)	`rs` (5)	`rt` (5)	`immediate` (16)
`beq rs, rt, relative address`	000100	`rs`	`rt`	`number of words`
`bne rs, rt, relative address`	000101	`rs`	`rt`	`number of words`
`addi rt, rs, immediate`	001000	`rs`	`rt`	`immediate`
`andi rt, rs, immediate`	001100	`rs`	`rt`	`immediate`
`ori rt, rs, immediate`	001101	`rs`	`rt`	`immediate`
`xori rt, rs, immediate`	001110	`rs`	`rt`	`immediate`
`lui rt, immediate`	001111	00000	`rt`	`immediate`
`lb rt, immediate(rs)`	100000	`rs`	`rt`	`immediate`
`lw rt, immediate(rs)`	100011	`rs`	`rt`	`immediate`
`sb rt, immediate(rs)`	101000	`rs`	`rt`	`immediate`
`sw rt, immediate(rs)`	101011	`rs`	`rt`	`immediate`

> **Warning**
>
> The number of words in the branch instruction is measured relative to `PC + 4`. That is, we jump to `(PC + 4) + (Immediate * 4)` if the branch is taken.

J-Format

The memory address is always 32 bits. However, since it must be well-aligned with offsets as multiples of 4, the last 2 bits can be ignored.

Mnemonic	`opcode (6)`	`address (26)`
`j address`	000010	26-bit target address (shifted left by 2 when used)

> ⚠ **Warning**
>
> 1. The full address is formed using the upper 4 bits of `PC + 4`. This can cause jump instructions to fail if `PC` is near a boundary—specifically, when the upper 4 bits of `PC + 4` differ from those of `PC`.
> 2. The maximum jump range in bytes is 2^{28} from `PC + 4`. In general, it follows the formula: $2^{\text{immediate}} \times \text{word size}$ (4 in MIPS).

Instruction Set Architecture (ISA)

Maximum and Minimum Number of Instructions

Case 1: There is at least 1 instruction on each instruction type.

> ☰ **Example**
>
> There are three types of instructions: A (4-bit opcode), B (7-bit opcode), and C (8-bit opcode). Find the maximum and minimum total number of instructions.
>
> Solution:
>
> 1. Max = $(2^8 - 2^{8-7} - 2^{8-4}) + (1) + (1) = 240$ (Maximize C / Minimize A and B)
> 2. Min = $(2^4 - 1) + (2^{7-4} - 1) + (2^{8-7}) = 24$ (Maximize A and B / Minimize C)
>
> In this example, we don't subtract 1 for the last instruction (C) since we don't need to allocate anything for the subsequent instruction types.

Case 2: Each instruction type has a minimum required number of instructions, which may vary, but some must be greater than 1.

> ☰ **Example**
>
> There are three types of instructions: X (2-bit opcode), Y (4-bit opcode), and Z (7-bit opcode). We need at least 2 X-Type and Y-Type Instruction with at least 1 Z-Type

Instruction. Find the maximum and minimum total number of instructions.

Solution: We use a similar approach as above but scale the relevant terms based on the number of instructions allocated for each instruction type.

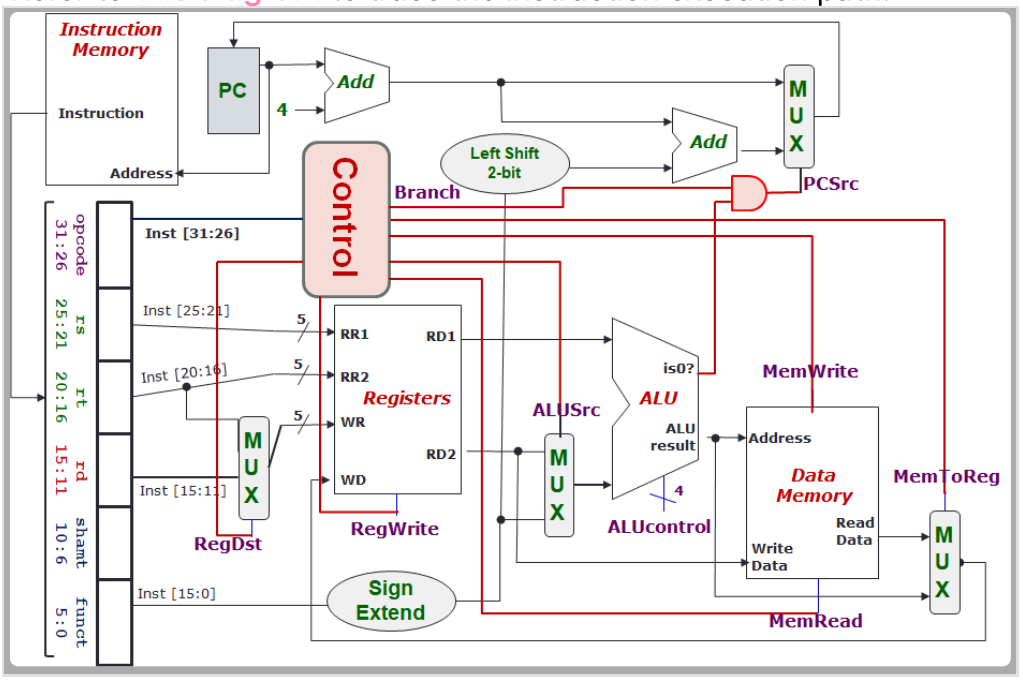
1. Max = $(2^7 - 2^{7-4}(2) - 2^{7-2}(2)) + (2) + (2) = 48 + 2 + 2 = 50$.
2. Min = $(2 + (2^2/2 - 1)) + (2 + (2^{4-2}/2 - 1)) + (2^{7-4}) = 3 + 3 + 8 = 14$.

In this example, we need to allocate two instructions for X and Y. Note that the minimum number of instructions does not need to be a power of two.

MIPS Datapath and Control

General Path

Refer to this diagram to trace the instruction execution path.



Info

The multiplexer `MemToReg` is reversed only because the wires cross on the diagram.

Tip

The standard control signals for different types of instruction are shown below.

Instruction	RegDst	ALUSrc	MemToReg	RegWrite
`R-type`	1	0	0	1
`lw`	0	1	1	1
`sw`	X	1	X	0

Instruction	RegDst	ALUSrc	MemToReg	RegWrite
`beq`	X	0	X	0

Instruction	MemRead	MemWrite	Branch	ALUop1	ALUOp0
`R-type`	0	0	0	1	0
`lw`	1	0	0	0	0
`sw`	0	1	0	0	0
`beq`	0	0	1	0	1

Critical Path

Example

Given below are the resource latencies of various hardware components in picoseconds (ps): Inst-Mem (400 ps), Adder (100 ps), MUX (30 ps), ALU (120 ps), Reg-File (200 ps), Data-Mem (350 ps), Control/ALU Control (100 ps), Left-shift/Sign-Extend/AND (20 ps). Determine the latency for the instruction `lw $24, 0($15)`.

Solution

- Fetch stage: Fetching the instruction from memory takes 400 ps. In parallel, `PC + 4` is computed using an adder, costing 100 ps, but this is not critical.
- Decode stage:
  - Reading the opcode to determine the instruction type and field lengths takes no time.
  - Reading data from the register file takes 200 ps.
  - The control unit determines control signals and propagates them in 100 ps, but this is not critical.
  - MUX inputs are pre-determined, so the `RegDst` and `ALUSrc` MUX takes no additional time later. However, other MUXs still need to wait for the input.
- ALU stage: Computing the memory address using the ALU takes 120 ps.
- Memory stage: Reading data from memory takes 350 ps.
- Register write stage: The result passes through the `MemToReg` MUX and is written to the register file, taking 30 + 200 = 230 ps.

Total latency: $400 + 200 + 120 + 350 + 230 = 1300$ ps.

Warning

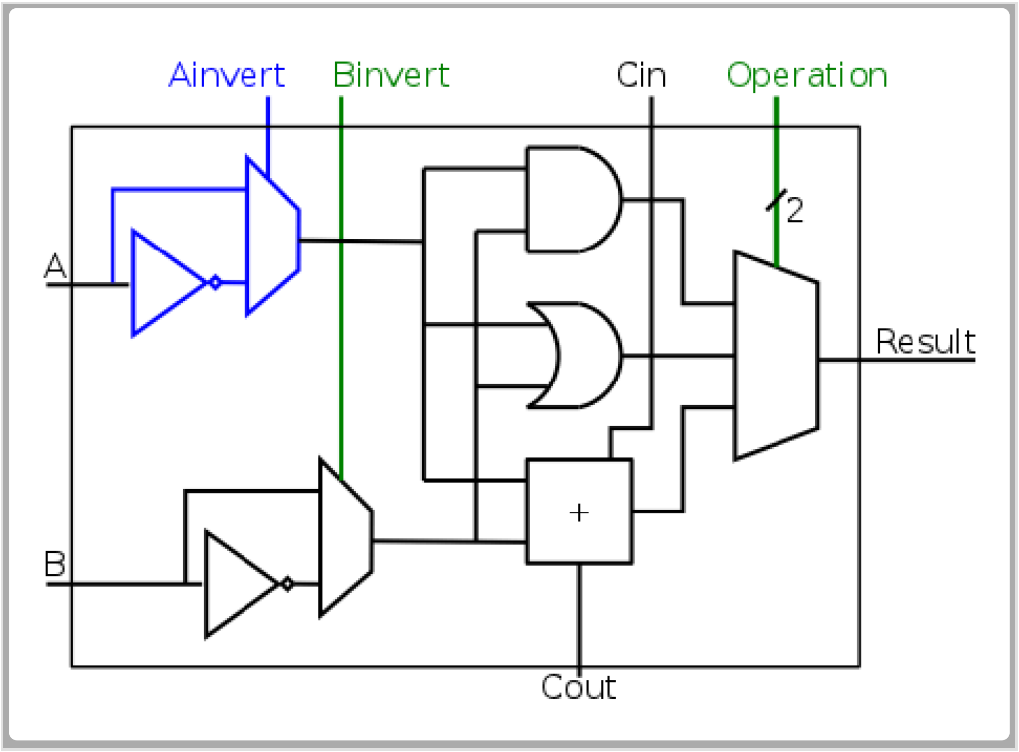
For branch instructions, there is a parallel execution of steps 2 and 3, which involve the following:

1. Sign-extending the immediate (20 ps),
2. Left-shifting by 2 (20 ps),
3. Adding to `PC + 4` (100 ps).

This combined operation costs a total of 140 ps, which is less than the combined latency of steps 2 and 3 in the above example.

After execution, the branch instruction waits at the `PCSrc` MUX for the `is0?` signal from the ALU, which is ANDed with the branch signal (20 ps) before passing through the MUX (30 ps).

ALU Slice



Tip

The standard ALUControl signals for different types of instruction are shown below.

Instruction	ALUControl
`lw`	`0010`
`sw`	`0010`
`beq`	`0110`
`add`	`0010`

Instruction	ALUControl
`sub`	`0110`
`and`	`0000`
`or`	`0001`
`slt`	`0111`

Note:

- The first two bits indicate A inverse and B inverse. B inverse is `1` only for subtraction.
- The last two bits follow the ALU slice's operation order: `00` for `and`, `01` for `or`, `10` for `add`, and `11` for `slt` (hidden).

Example

Given that all logic gates take 1 ps (picosecond) and MUXs take 2 ps, determine the maximum latency of a 4-bit ALU.

Solution:

- Inputs A and B arrive in parallel. The longest delay comes from inverting both, which takes $\max\{1+2, 1+2\} = 3$ ps.
- All operation gates also run in parallel, taking $\max\{1, 1, 1\} = 1$ ps for bit 0. However, carry propagation occurs from LSB to MSB. Since all slices operate in parallel, bits 1, 2, and 3 must wait 1 ps per previous bit. This delay accumulates, making the critical path for the MSB take $1 + 1 + 1 = 3$ ps. Note that each 1 corresponds to the operation gate, not the propagation.
- The total time so far is 6 ps. After passing through the operation MUX, the final delay is $6 + 2 = 8$ ps.

Good to Memorize

- No Operation (NOP) can be implemented by an instruction that avoids reading/writing to memory or modifying registers.
- Register File is a set of 32 registers, excluding immediate values.
- Instruction Register (IR) holds the encoded instruction currently being executed.
- Special Register: The stack pointer (`$sp`) points to the last occupied location at the top of the stack, which grows downward in memory.
- Rising Edge of the Clock Cycle: The moment when the program counter (`PC`) is updated.

- Single-Cycle Implementation: The cycle time is determined by the slowest instruction.
- Multi-Cycle Implementation: Each instruction is broken into steps, with each step taking one cycle. The overall cycle time depends on the slowest step.
- Implementation of `slt`: We get the sign bit from bit 31 and carry that to be bit as well as setting the remaining bits to be 0. If the result is negative, bit 0 will be 1.
- Endianness refers to the order in which bytes are arranged in a multi-byte word stored in memory. In big-endian format, the most significant byte (MSB) is stored at the lowest memory address, while in little-endian format, the least significant byte (LSB) is stored at the lowest memory address.

1 - Combinational Circuits

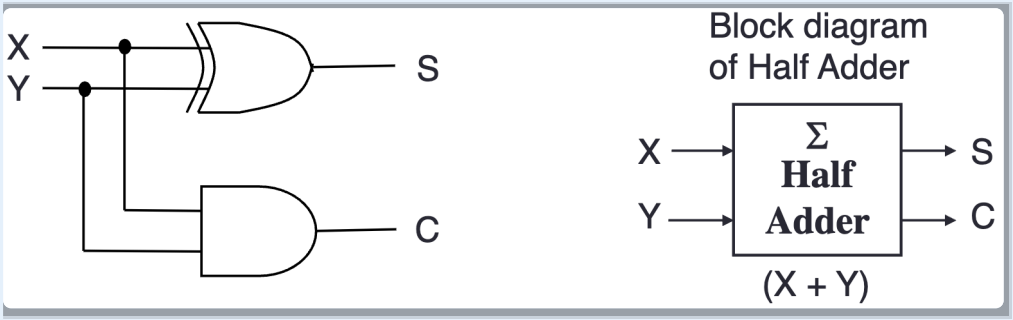
Introduction

- Combinational Circuit: Output depends only on the input.
- Sequential Circuit: Output depends on both input and state (can vary for the same input).

Gate-Level (SSI) Design

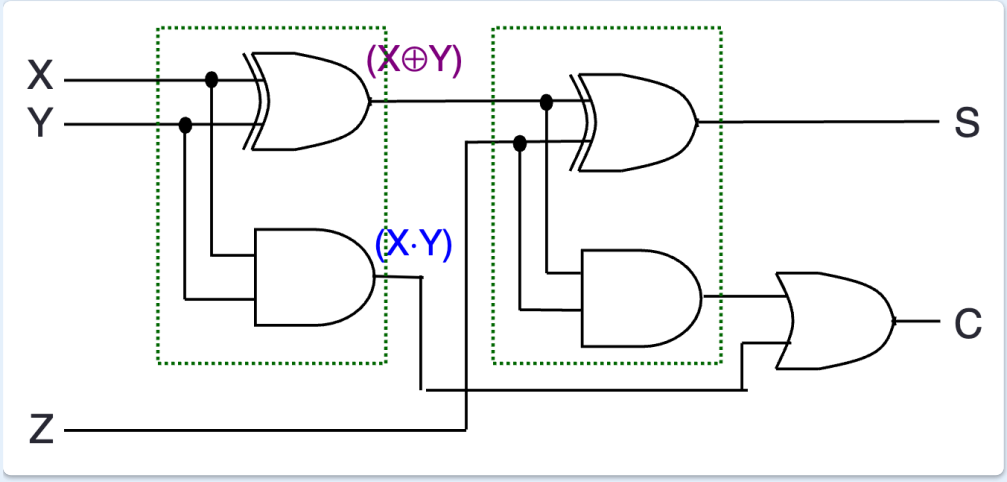
Half Adder

A half adder takes two inputs, X and Y, and produces two outputs: C (Carry Out) and S (Sum).

> **Info**
>
> From the truth table:
>
> - $C = X \cdot Y$
> - $S = X \oplus Y$
>
> | X | Y | C | S |
> |---|---|---|---|
> | 0 | 0 | 0 | 0 |
> | 0 | 1 | 0 | 1 |
> | 1 | 0 | 0 | 1 |
> | 1 | 1 | 1 | 0 |
>
> Logic diagram:
>
> 
>

Full Adder (FA)

A full adder extends the half adder by taking three inputs: X, Y, and Z. Here, Z represents Carry In, which allows proper binary addition.

> **Note**
>
> Using K-maps, we simplify the expressions:
>
> - $C = X \cdot Y + (X \oplus Y) \cdot Z$
> - $S = X \oplus (Y \oplus Z)$
>
> Logic diagram:
>
> 
>
>
> Remark: The bordered section of the diagram consists of two half adders, which is why this is called a full adder.

> **Success**
>
> To derive the expressions, note these XOR properties:
>
> 1. $X + Y = (X \oplus Y) + (X \cdot Y)$
> 2. $X \oplus Y = X' \cdot Y + X \cdot Y'$
> 3. $(X \oplus Y)' = X' \cdot Y' + X \cdot Y$

Block-Level Design

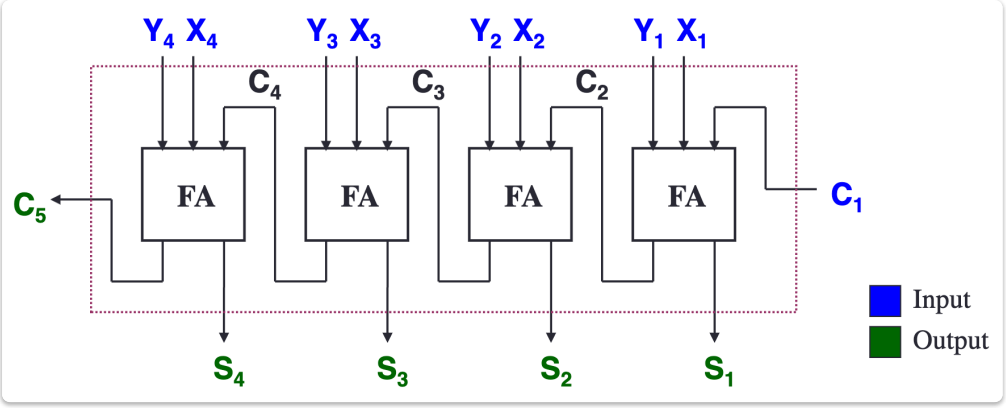
4-bit Parallel Adder

Adding 4-bit binary numbers requires large truth tables to derive expressions. However, we observe that: $C_{i+1}S_i = X_i + Y_i + C_i$. This can be represented using a sequence of full adders.

> **Tip**
>
> Using full adders, for bit i, we simplify the expression as:
>
> - $C_{i+1} = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_i$

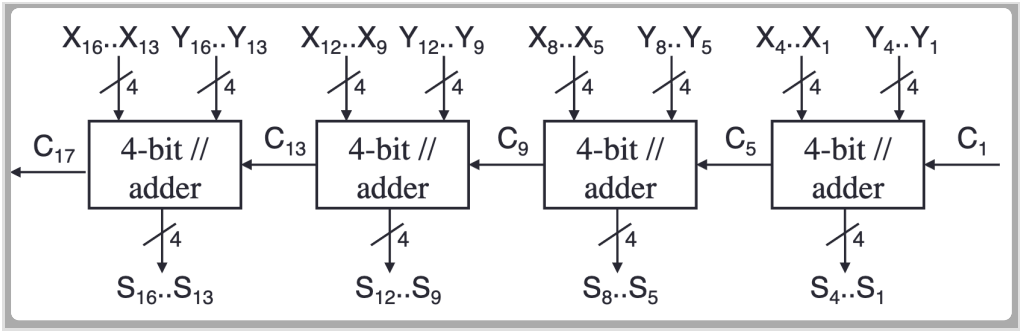
- $S_i = X_i \oplus Y_i \oplus C_i$

Thus, we get the following circuit diagram, known as a ripple-carry adder:



Applications of Parallel Adder

- BCD to Excess-3 Converter: To convert a number to Excess-N format, add N to it. This can be implemented using an adder circuit.
- 16-bit Parallel Adder: Built using a series of 4-bit parallel adders.



Magnitude Comparator

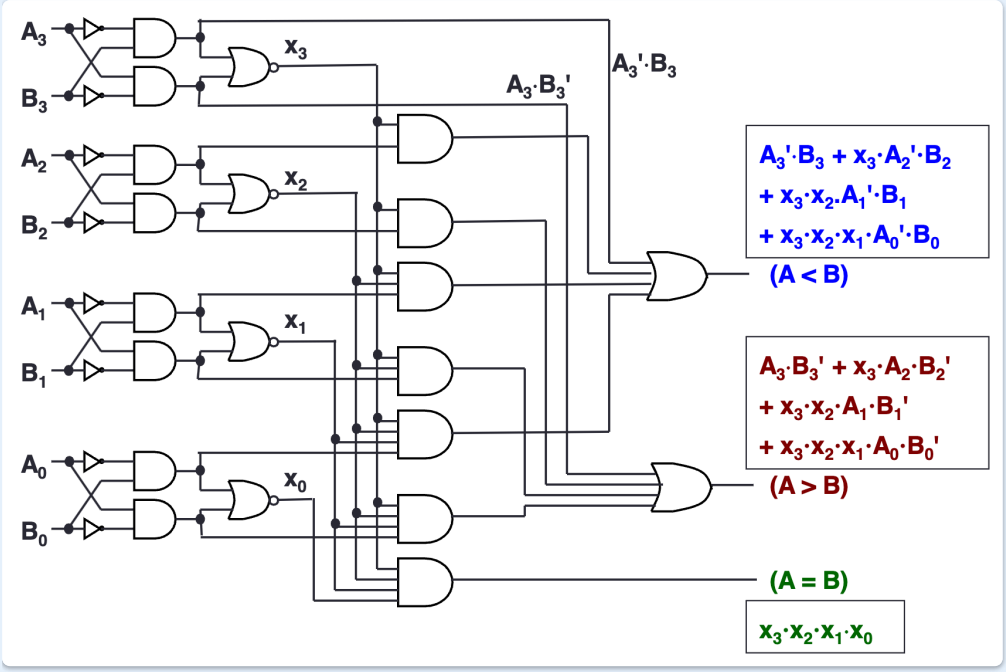
For n-bit unsigned values, compare bits from MSB to LSB. If the result is undecided, continue until the LSB.

> **Tip**
>
> To check if two bits are equal, use:
>
> $$x_i = A_i \cdot B_i + A_i' \cdot B_i'.$$
>
> This expression is 1 when both A_i and B_i are either 0 or 1.

> **Note**
>
> For a 1-bit comparison:

- $A_i < B_i$ when $A_i' \cdot B_i$
- $A_i > B_i$ when $A_i \cdot B_i'$

The circuit follows this logic:



Remark: Before checking lower bits, ensure all upper bits are equal using AND with x_i .

We can further apply multiple comparator with AND/OR to check a two-way inequalities like $X < Y < Z$ or $X > Y$ and $Y < Z$ conditions.

Circuit Delays

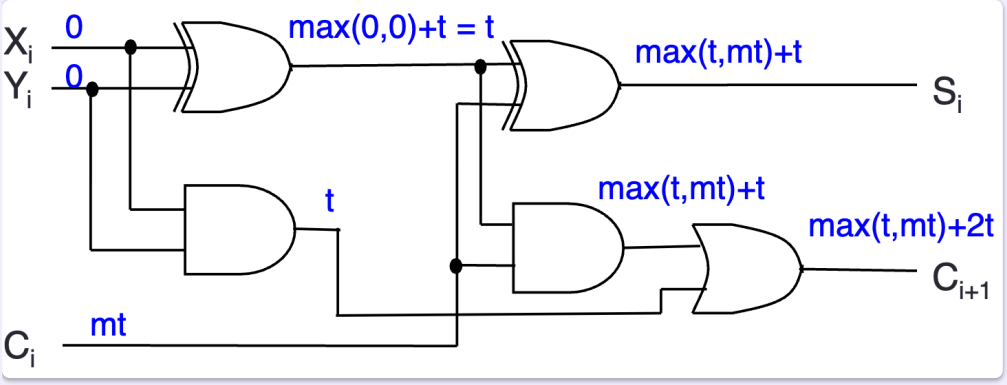
We assume all logic gates have the same delay, denoted by t. If the inputs to a gate arrive at times $t_1, t_2, \ldots, t_n$, the output becomes ready at:

$$\max\{t_1, t_2, \ldots, t_n\} + t.$$

Example

Consider the delay of a full adder (FA) in a parallel adder:

- All bit inputs arrive at time 0.
- There is a delay before the carry out from previous bits becomes available.



1. For $i = 1, m = 0$:
   - S_1 is ready at $2t$
   - C_2 is ready at $3t$.
2. For $i = 2, m = 3$:
   - S_2 is ready at $4t$
   - C_3 is ready at $5t$.
3. For $i = 3, m = 5$:
   - S_3 is ready at $6t$
   - C_4 is ready at $7t$.

In general, for an n-bit ripple carry adder:

- The delay for S_n is $((n-1) \times 2 + 2)t$.
- The delay for C_n is $((n-1) \times 2 + 3)t$.

The overall maximum delay is determined by C_n.

2 - MSI Components

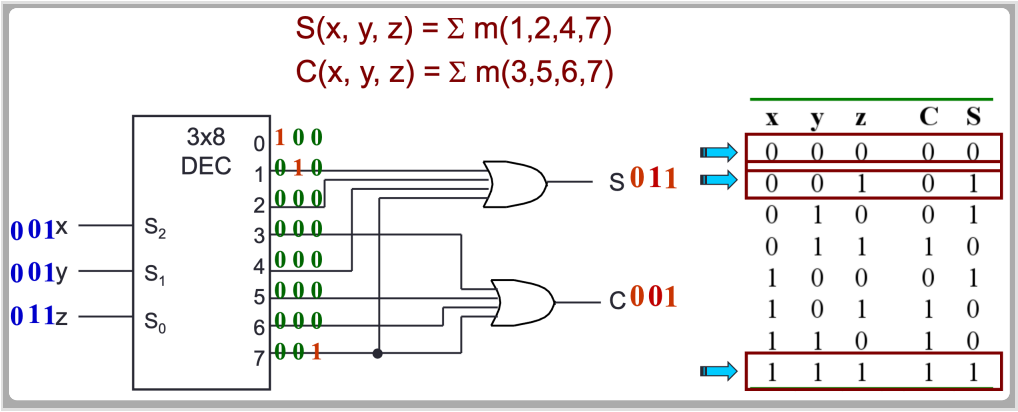
Introduction

Medium-scale Integration (MSI) refers to an Integrated Circuit (IC) containing 500–20,000 transistors and 100–9,999 logic gates.

Decoders and Encoders

Decoders (DEC)

Decoders convert n inputs into up to 2^n outputs. They are used to implement functions by connecting their outputs to logic gates. The design is based on the sum of minterms or product of maxterms expressions.



> **Note**
>
> Types of Decoders
>
> 1. Normal Decoder: Notation $n : 2^n$ or $n \times 2^n$.
> 2. 0/1-Enabled Decoder: Includes an extra enable signal that activates outputs when the signal is 0 or 1, denoted with $\bar{E}$ or E.
> 3. Negated Decoder: Uses active-low outputs where 0 indicates active and 1 indicates inactive.

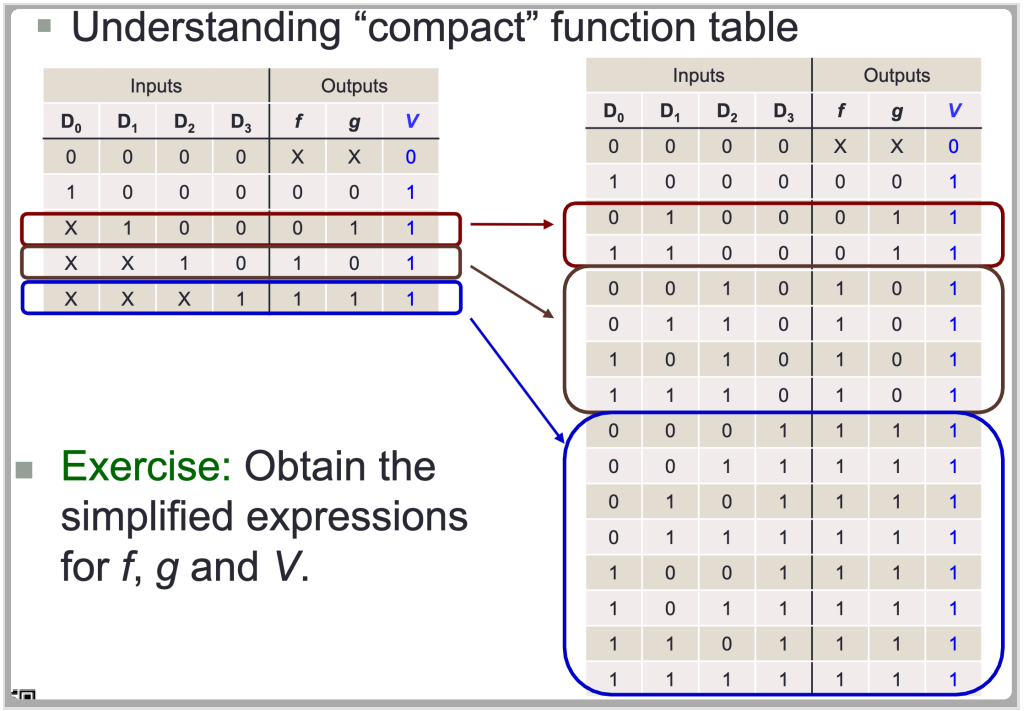
> **Example**
>
> Function Implementation:
>
> $$f(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$$
>
> Solutions:

- **Active-High Outputs with OR Gate:** $f(Q, X, P) = m_0 + m_1 + m_4 + m_6 + m_7$
- **Active-Low Outputs with NAND Gate:** $f(Q, X, P) = (m_0' \cdot m_1' \cdot m_4' \cdot m_6' \cdot m_7')'$
- **Active-High Outputs with NOR Gate:** $f(Q, X, P) = (m_2 + m_3 + m_5)'$ (Equivalent to $M_2 \cdot M_3 \cdot M_5$)
- **Active-Low Outputs with AND Gate:** $f(Q, X, P) = m_2' \cdot m_3' \cdot m_5'$

Encoders (ENC)

Encoders perform the reverse operation: they take in up to 2^n inputs and produce n output bits. Their design can be approached via **K-maps or logical observation**.

> **Info**
>
> **Types of Encoders**
>
> | Characteristic | Normal Encoder | Priority Encoder |
> |---|---|---|
> | Inputs | Only one input high at a time | Multiple inputs may be high; selects the highest priority input |
> | Outputs | Invalid inputs yield "don't care" outputs | Returns a coded output where non-selected inputs are treated as 0 |
>
> **Note:** Priority encoders benefit from more compact truth tables.

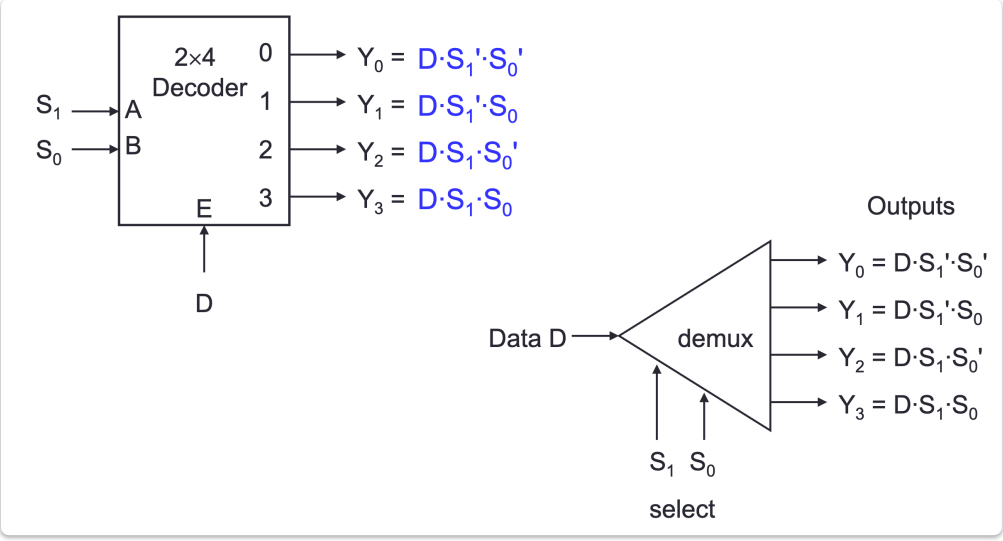


Demultiplexers and Multiplexers

A **multiplexer (MUX)** selects **one input source** to pass to the output, while a **demultiplexer (DEMUX)** directs the input to **one of the multiple destinations**.

Demultiplexers (DEMUX)

DEMUX functions similarly to a decoder with the data acting as the enable signal and the destination selection as inputs.



Multiplexers (MUX)

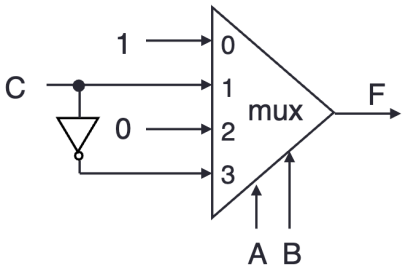
A MUX has 2^n inputs and n selection lines, producing a single output based on the selection lines (similar to an encoder). Notation: $2^n : 1$ MUX.

> **Tip**
>
> Output Representation of a 4-to-1 MUX:
>
> $$I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$$
>
> Alternatively:
>
> $$I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3.$$

> **Example**
>
> Implementing Functions with a Small MUX
>
> Given:
>
> $$F(A, B, C) = \sum m(0, 1, 3, 6)$$
>
> Using an 8-to-1 MUX, we could set $I_1, I_3, I_5, I_6 = 1$ and others to 0.
> But can we optimize?
>
> Solution:
>
> 1. Reserve one variable for data inputs, using the remaining for selection.
> 2. In this case, choose C for data (least significant), and A, B for selection.

3. Analyze the truth table to reconstruct the circuit efficiently.

A	B	C	F	MUX input
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	C'
1	1	1	0	



3 - Sequential Circuits

Introduction

Sequential Circuit = Combinational Circuit + Memory State. The outputs depend on both inputs and the memory state.

Memory Elements

Let $Q = Q(t)$ denotes the current state and $Q^+ = Q(t+1)$ denotes the next state.

1. Set: $Q^+ = 1$
2. Reset: $Q^+ = 0$
3. Memorise: $Q^+ = Q$

The update in states is determined by clock (square waves).

Latches

Latches are triggered by pulses (ON = 1, OFF = 0). It give two complement outputs: Q and Q'. Both helps in connecting circuits efficiently, reducing extra inverters.

> **Info**
>
> Different Types of Latches
>
> | Latch | Input | Set | Reset | Memorise | Invalid |
> | --- | --- | --- | --- | --- | --- |
> | S-R | SR | $SR = 10$ | $SR = 01$ | $SR = 00$ | $SR = 11$ |
> | Gated D | $D = SS'$ | $EN = 1, D = 1$ | $EN = 1, D = 0$ | $EN = 0$ | - |
>
> Remark: Gated D use EN for enabling the signal and use $R = S'$.

Flip-flops

Flip-flops are triggered by edges (Positive: ON = from 0 to 1, Negative: ON from 1 to 0 and OFF otherwise). The clock signal that enables them is denoted by $\uparrow$ or $\downarrow$.

> **Info**
>
> Different Types of Flip-flops

Flip-flops	Input	Set	Reset	Memorise	Invalid	Toggle ($Q^+ = Q'$)
S-R	SR	$SR = 10$	$SR = 01$	$SR = 00$	$SR = 11$	-
D	$D = SS'$	$D = 1$	$D = 0$	$\downarrow$	-	-
J-K	JK	$JK = 10$	$JK = 01$	$JK = 00$	-	$JK = 11$
T	$JK = TT$	-	-	$T = 0$	-	$T = 1$

Remark: All are triggered when during $\uparrow$ (work like EN in latches).

Note

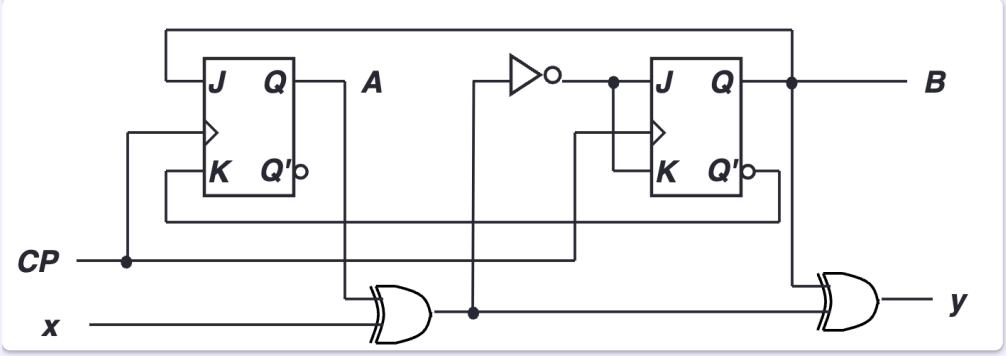
Both latches and flip-flips are example of synchronous inputs, the output changes at specific time. Note that asynchronous means outputs can change any time.

Sequential Circuits In Action

Analyzing Circuits

Example

Derive the state table and state diagram of this circuit.

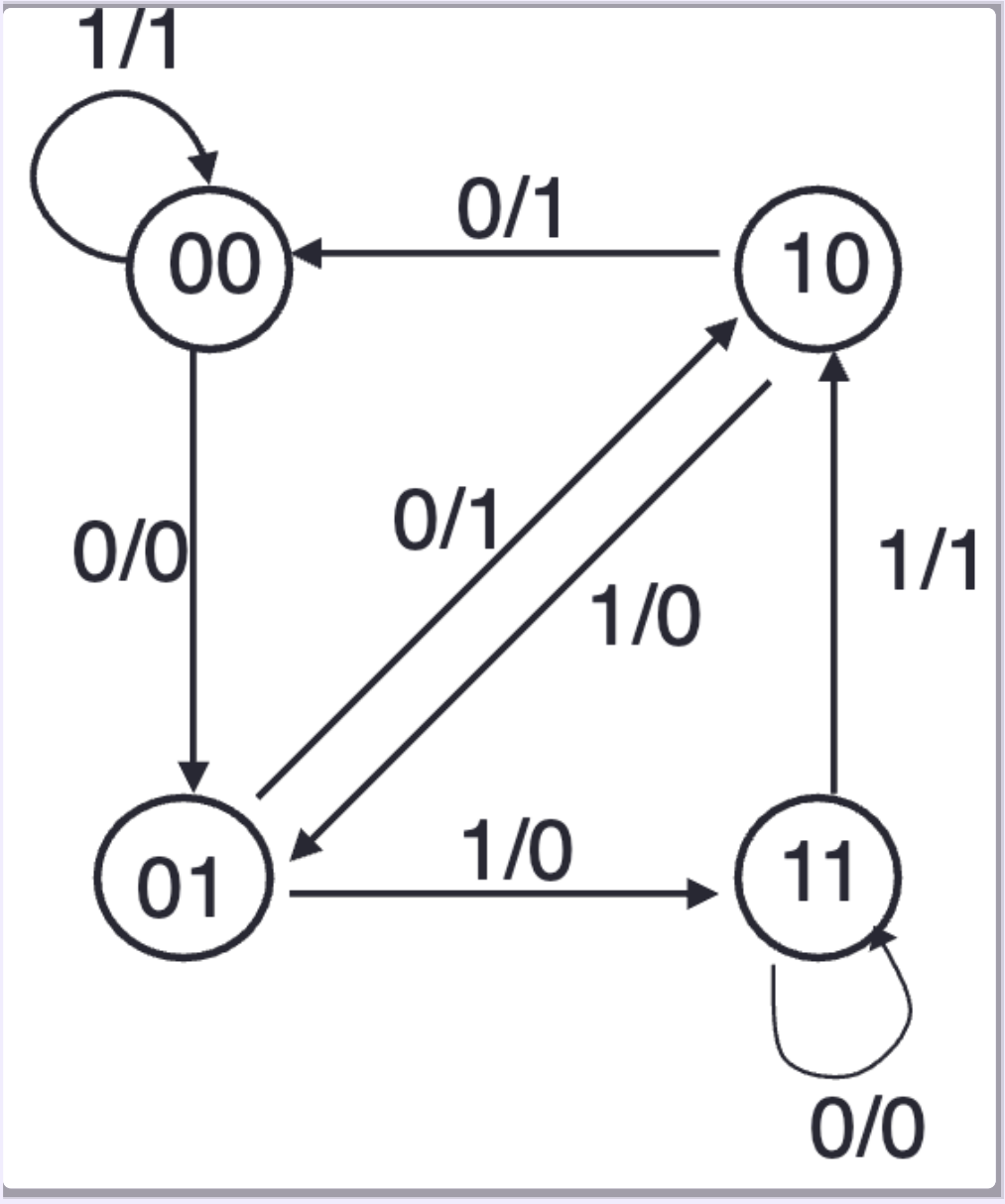


Solution:

1. First, we write down the flip-flop input functions: $JA = B$, $KA = B'$, $JB = (A \oplus x)' = A \cdot x + A' \cdot x' = KB$. Note that JA is the input J of the flip-flop A.
2. We can now write down the state table as followed, where $y = A \oplus x \oplus B$

Present state		Input	Next state		Output	Flip-flop inputs			
A	B	x	A^+	B^+	y	JA	KA	JB	KB
0	0	0	0	1	0	0	1	1	1
0	0	1	0	0	1	0	1	0	0
0	1	0	1	0	1	1	0	1	1
0	1	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	1	0	0
1	0	1	0	1	0	0	1	1	1
1	1	0	1	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1	1

3. Finally, we arrive at the following state diagram, where each node is AB and the label on each edge is x/y.



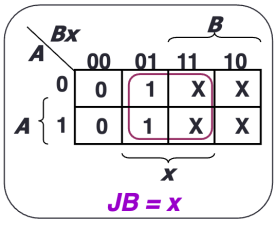
Designing Circuits

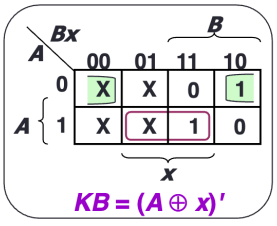
Example

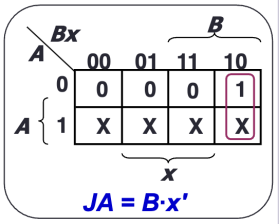
Derive the state equation from state diagram

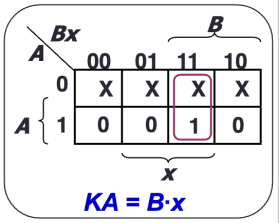
1. Write the state table consisting of the present state, input, and next state.
2. Use the excitation table to reverse engineer the flip-flop inputs.
3. Create K-maps for all flip-flop inputs using present states and inputs as the variables.

Present state		Input	Next state		Flip-flop inputs			
A	B	x	A^+	B^+	JA	KA	JB	KB
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1









Tip

We can quickly get the flip-flop input from this excitation table.

Q	Q^+	SR	JK	D	T
0	0	0X	0X	0	0
0	1	10	1X	1	1
1	0	01	X1	0	1
1	1	X0	X0	1	0

Memory

Memory Hierarchy

- Fastest: Registers > Main Memory > Disk Storage > Magnetic Tapes
- Largest in Size: Magnetic Tapes > Disk Storage > Main Memory > Registers

Note

There is a trade-off between speed and size. Below is the list of sizes:

- 1 KB = 2^{10} bytes
- 1 MB = 2^{20} bytes
- 1 GB = 2^{30} bytes
- 1 TB = 2^{40} bytes

Memory Operation

All operations are activated when memory is enabled (signal is 1).

- **Read/Write = 0**: Write to the selected word.
- **Read/Write = 1**: Read from the selected word.

> **Note**
>
> RAM is just memory array. Static RAMs use flip-flops as the memory cells while dynamic RAMs use capacitor charges, requiring constant refreshes.

4 - Pipelining

Introduction

- Pipelining is a technique that helps speed-up the entire workload (but not a single one).
- Steady State is the properties of the workload which is defined by the pipeline rate, i.e. it is looking at the state with maximum efficiency,

> **Key ideas**
>
> 1. Multiple Tasks operating simultaneously using different resources.
> 2. Pipeline rate is determined by the slowest stage
> 3. It could have possible delays due to stalling for dependencies.

MIPS Pipeline Stage

Core ideas

- Each stage takes 1 clock cycle.
- In general, the flow of the data is from one to the next (with some exceptions on updating PC and write back to register).
- We can pipeline the execution stages of multiple instructions!

> **Execution stages**
>
> - IF: Instruction Fetch
> - ID: Instruction Decode and Register Read
> - EX: Execute an operation or calculate an address
> - MEM: Access an operand in data memory
> - WB: Write back the result into a register

Pipeline Datapath

Since we are executing multiple instructions at the same time, we need registers called Pipeline Registers to keep track of the data used by same instruction in later pipeline stages.

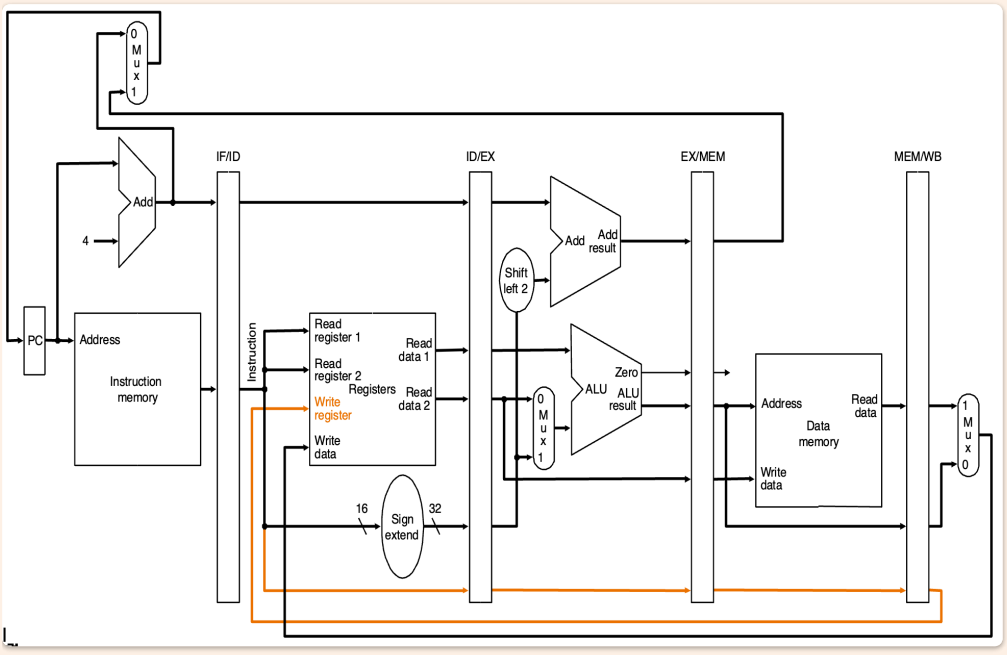
> **Pipeline datapath**

- IF: Instruction Read and $PC + 4 \rightarrow$ IF/ID $\rightarrow$ Register Numbers, 16-bit offset to be sign-extended, and $PC + 4$.
- ID: Data values from registers, 32-bit immediate, and $PC + 4 \rightarrow$ ID/EX $\rightarrow$ (same).
- EX: $(PC + 4) + (\text{Immediate} \times 4)$, `isZero?` signal, Data Read 2 $\rightarrow$ EX/MEM $\rightarrow$ (same).
- MEM: ALU result and Memory Read data $\rightarrow$ MEM/WB $\rightarrow$ (same).
- WB: At the end of the cycle, result is written back to register if applicable.

Remark: The values are shown as: value 1 $\rightarrow$ Pipeline Registers $\rightarrow$ value 2. Value 1 is the values stored before the start (end of the previous) and value 2 is that stored at the end (starting of the next).

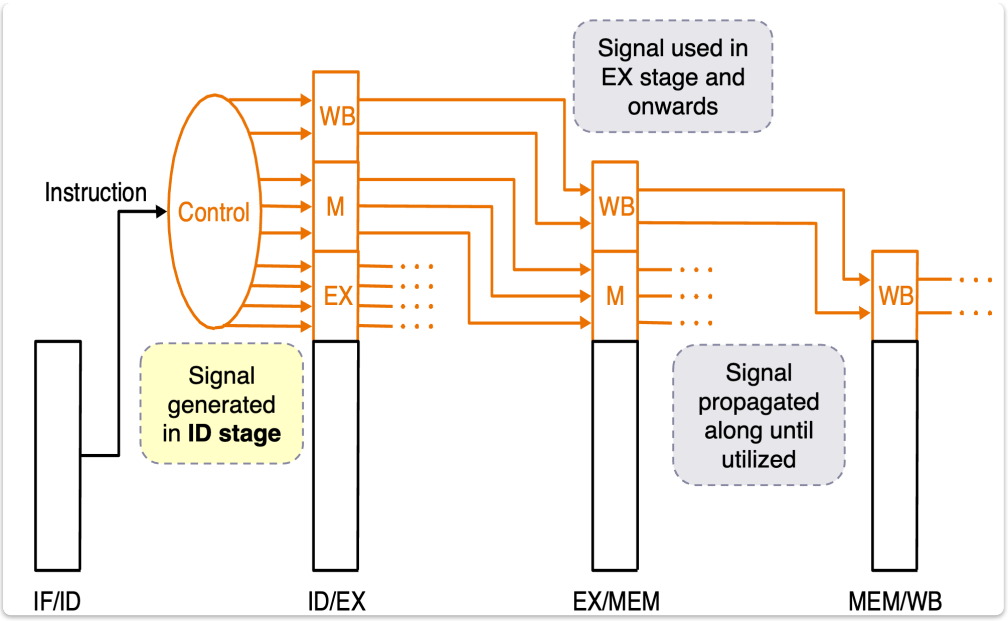
⚠ Warning

Notice that there is a potential bug here. The value of `WB` is not the same anymore after executing next instructions. We need to also pass the value of `WB` all the way to the end.



Pipeline Control

- We notice that control signals are used at certain stages. We can store these controls along with the pipeline registers.
- Some signals will not be stored anymore after each stage since they are already used.



Grouping control

- EX Stage: `RegDst`, `ALUSrc`, and `ALUop`.
- MEM Stage: `MemRead`, `MemWrite`, `Branch`.
- WB Stage: `MemToReg`, `RegWrite`.

Performance Comparisons

Implementation	Description	Cycle Time	Total Execution Time
Single-Cycle	One instruction per cycle	$CT = \max(\sum_{k=1}^{N} T_k)$	$I \times CT$
Multi-Cycle	One stage per cycle (resulting in shorter cycles)	$CT = \max(T_k)$	$I \times CT \times \text{Average CPI}$
Pipeline	One stage per cycle (but run in pipeline)	$CT = \max(T_k) + T_d$	$(N + I - 1) \times CT$

Remark

1. Total execution time is calculated for I instructions.
2. Cycles per instruction (CPI) is required since different instructions involved different number of stages and thus required different cycles.
3. T_d is the overhead in pipelining, such as reading and writing pipeline registers.

Success

Pipeline Speedup is the ideal performance gained, considering all instructions take the same time T, $I \gg N$, and $T_d = 0$, which is

$$\frac{I \times N \times T}{(N + I - 1) \times T} \approx \frac{I \times N \times T}{I \times T} = N \text{ times speed up.}$$

Pipeline Hazards

Different Types of Hazards

Speedup is based on the assumption that a new instruction could be pumped into pipeline every cycle. In reality, there are some problems that prevent these from happening:

1. Structural Hazards: Using the same hardware resource at the same time.
2. Data Hazards: Data dependencies between instructions (`R/W` from the same register).
3. Control Hazards: Change in the program flow, e.g. `beq` .

Structural Hazards

Conflicts on Memory

1. Use Stalling: Delay the later instruction until there is no overlap.
2. Use Two Memories: Split the memory into Data and Instruction. Loading the data will use the former while reading instructions will use the latter.

Conflicts on Register

By exploiting the fact that registers are really fast memory. We can do `RegWrite` in the first half of the cycle and do register reading in the second half.

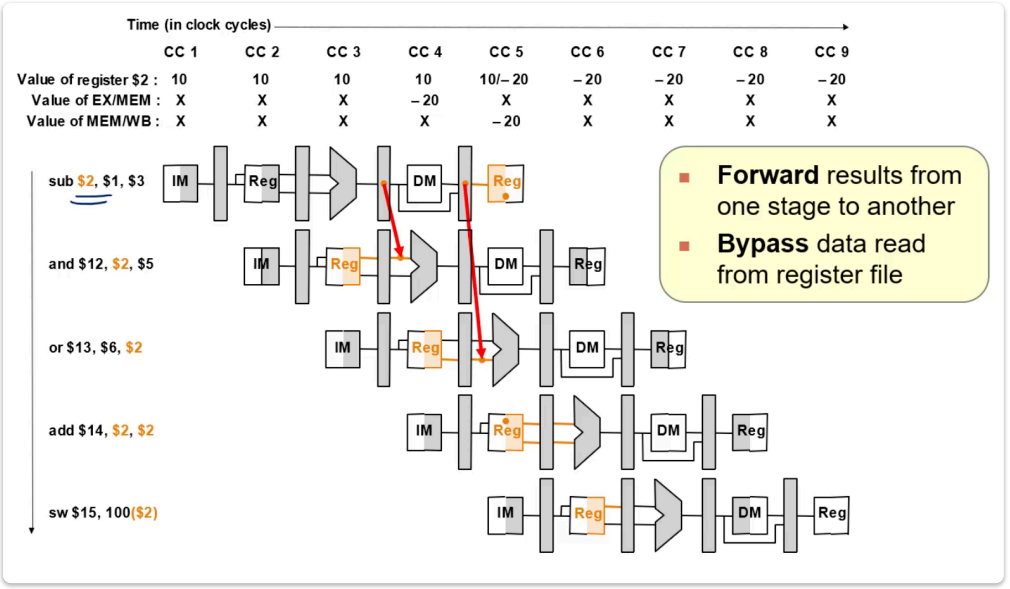
> ⚠ Warning
>
> It is always the case to write first, following the order of instruction (write is done in the last stage of the first instruction, if applicable).

Data Hazards

- Read-After-Write (RAW) is the data dependency that could cause the problem (but not the WAR or WAW).
- This is when the data will be written in register in later cycle but we need to read from registers when executing subsequent instructions before that.

Forwarding

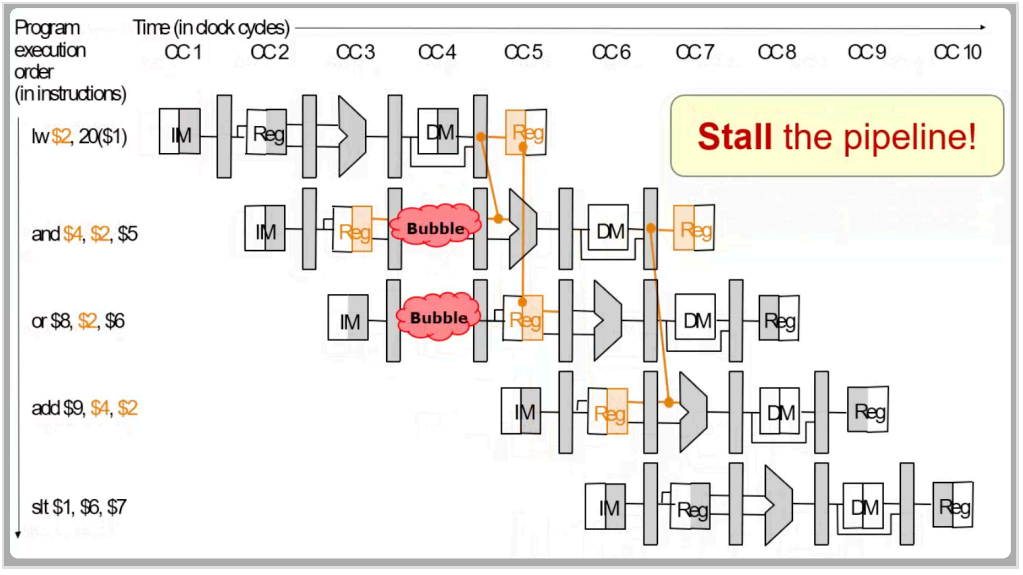
Since the data read from registers will be used for ALU computation. We can forward the data as soon as it is ready to the next instruction without waiting for `RegWrite`.



> **Remark**
>
> There is no need to forward to some later instructions after the `RegWrite` since the register will already be available at that time.

Load Instruction

In this case, the written register is only available after reading from memory. In this case, we also need to stall the pipeline before forwarding.

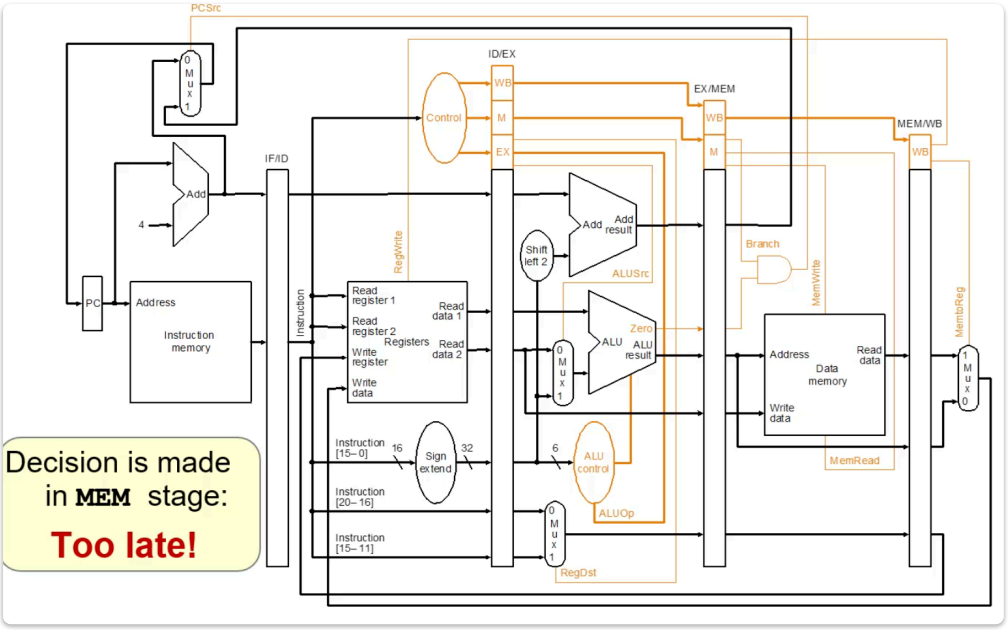


> **Tip**

Notice that forwarding and stalling is not necessary when register is read and written in the same clock cycle. The first instruction is execute before the latter by default.

Control Harzards

The branch instruction which decides which instructions to execute are only available in the MEM stage which is after 3 clock cycles.



One quick solution is to add stalls for 3 clock cycles but this could be huge penalties since modern programs have lots of these instructions.

Early-Branch Technique

We add extra computational unit in the ID stage to get the branch (without waiting for ALU). This generally creates only a stall of 1 clock cycle. However, if the registers used in branching have data dependencies, we can still experience 3 clock cycles.

Branch Prediction Technique

Together with the previous technique, we can just assume that the branch is not taken, this will lead to executing some stages in advanced. If the prediction is correct, we save some stalls. If it is wrong, there is no significant improvement.

Delayed Branching Technique

We observe that using the previous techniques will leave us some X numbers of stalls ($X = 3$ in early branching and $X = 1$ in branch prediction). If there are some instructions to be executed whether or not the branch would be taken and also have no data dependency, the compiler can replace the stalls with them!

1. **Best-Case:** There exists **enough instructions** for us to do such technique.
2. **Worst-Case:** We use NOP (No Operations) in place of the stalls.

Pipelining Hacks

Possible Cases	Dependency	Without Forwarding	With Forwarding
Not Branching	RAW After Non-`lw`	+2	+0
	RAW After `lw`	+2	+1
Branching (Late/Early)	NA	+3/+1	NA
	RAW After Non-`lw`	+5/+3	+3/+2
	RAW After `lw`	+5/+3	+4/+3
Jump (Late/Early)	NA	+3/+1	NA

Additional Notes

1. Delay from RAW could be reduced by adding instruction without dependency in-between (could be complicated; safer to do with diagram).
2. Branch prediction (not taken) will reduced branching delay to 0 if the prediction is correct.

5 - Cache

Basic Ideas

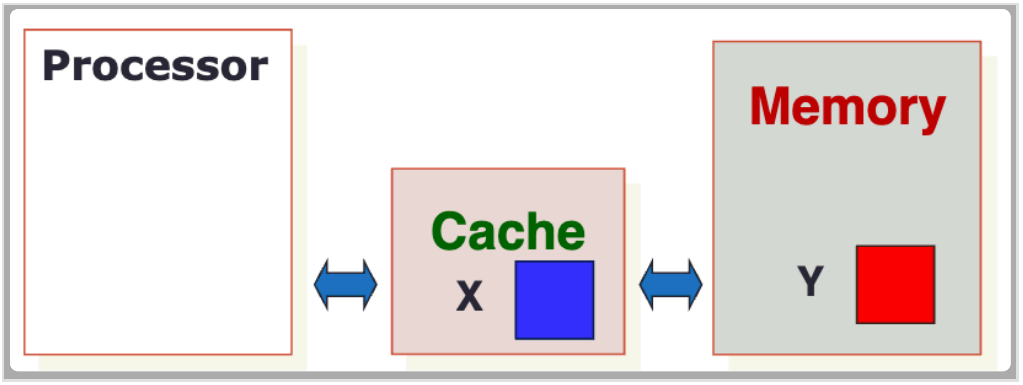
Principle of Locality

- We want a way to access bigger memory faster.
- Suppose, we store data x, we can adopt the principle of locality to store something similar to x since we are likely to need them soon.

> **Info**
>
> 1. Temporal Locality: If x is referenced at t, we might need it again at $t + \Delta t$.
> 2. Spatial Locality: If x is referenced, we might also need $x \pm \Delta x$.

Memory Access Time

Cache memory is SRAM (faster) while normal memory is DRAM (slower). We put cache between processor and memory.



> **Terminology**
>
> 1. Hit: We need X.
> 2. Miss: We need Y.

> **Example**
>
> Suppose our on-chip SRAM (cache) has 0.8 ns access time, but the fastest DRAM (main memory) we can get has an access time of 10 ns. How high a hit rate do we need to sustain an average access time of 1 ns?

Solution: $1 = 0.8 \times \text{hit rate} + (1 - \text{hit rate}) \times (10 + 0.8)$. Hence, we need a hit rate of 98%.

Remark: The above calculation is simply weighted average. If we miss, the required time is both accessing the memory and the cache.

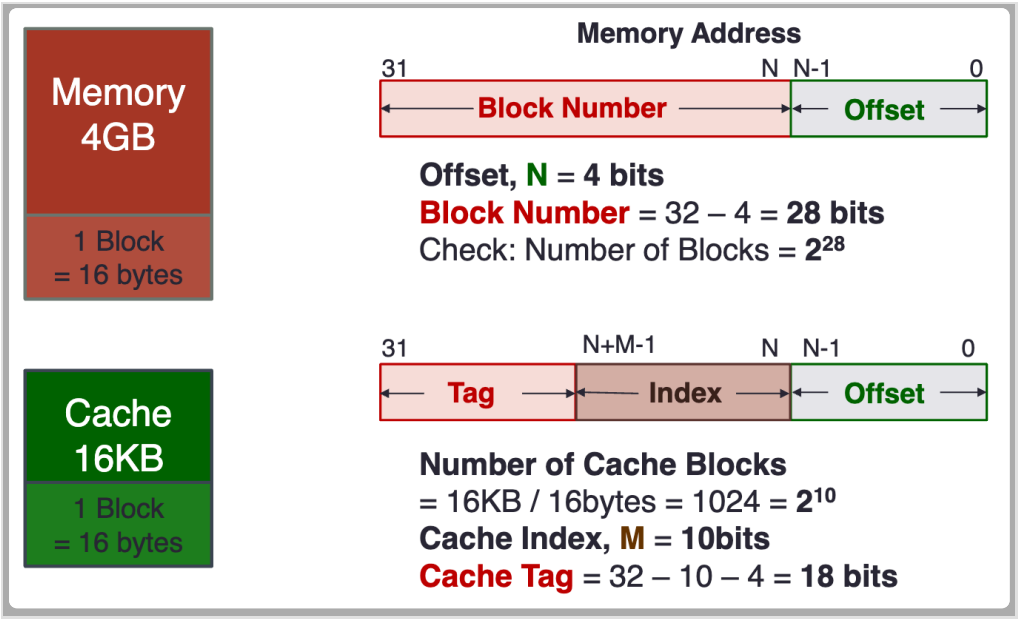
Direct Caching

Core Concepts

Cache Block is the unit of transfer between memory and cache. For example, 16-byte block consists of 4 words.

✓ Success

1. We made an observation that the first $32 - N$ bits of the 2^N-byte block are identical.
2. We call bit $31 : N$ as block number and $N - 1 : 0$ as byte offset.
3. The last M bits of the block number is the cache index: `Block Number % Number of Blocks`.
4. Many blocks have same index. We need a tag: `Block Number / Number of Cache Blocks`.

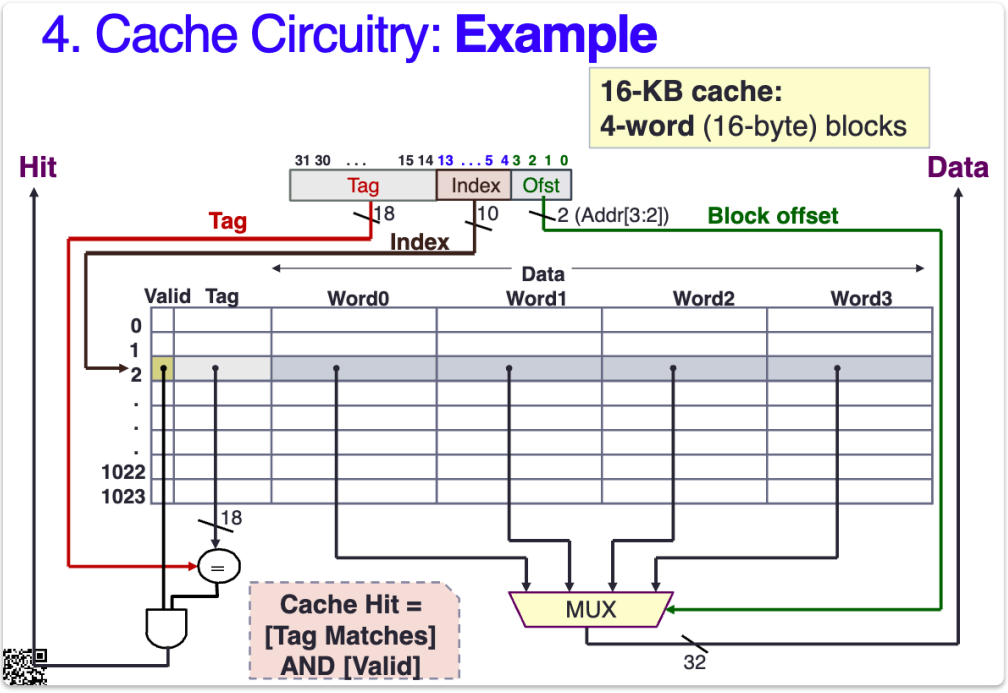


Remark:

1. Cache block need to store the same amount of data as memory block, e.g. 16 bytes.
2. Since the address is word-aligned. We can ignore the last two bits of the offset.

Cache Structure

1. We maintain two overheads: tag of the memory block, and valid bit (boolean) to indicate hit and miss.
2. There is a hit when the valid is 1 and `Tag[Index] = Tag[Memory Address]`.
3. If it is a miss, we load the tag and 16 bytes of the memory (simply replace the data when loading different tag to the same index).
4. In either case, return the word at offset to the register.

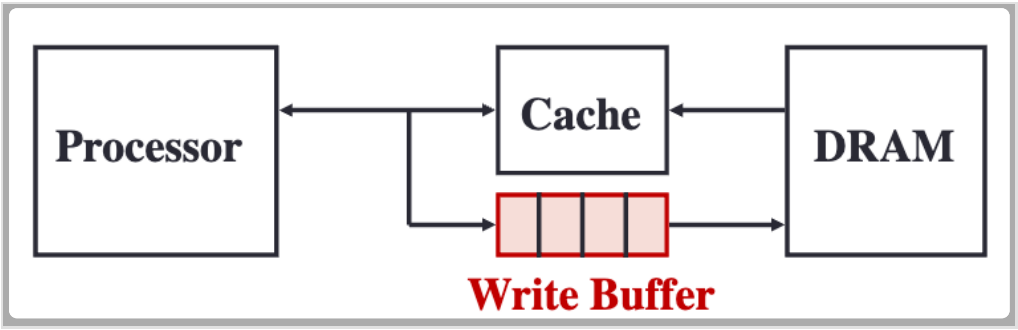


Types of Cache Misses

1. Compulsory (Cold): First access of the block.
2. Conflict: Mapped to the same index.
3. Capacity: Cache cannot contain all blocks needed (applied to fully associative cache).

Write Policy

Writing on cache and cause inconsistency between data in cache and memory.



1. Processor writes data to cache and buffer.
2. Memory controller write contents of the buffer to the memory.
3. Maintain another bit (dirty bit). Write operation will change this bit to 1.

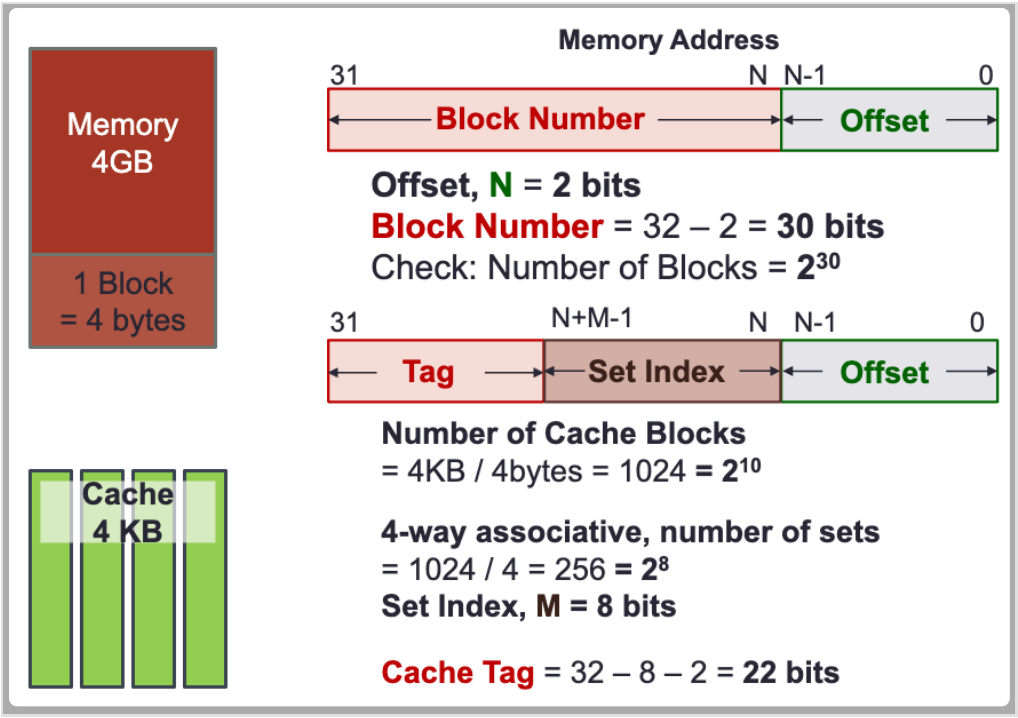
4. When a cache block is replaced (during read), write back to memory if dirty bit is 1.

Set Associative Caching

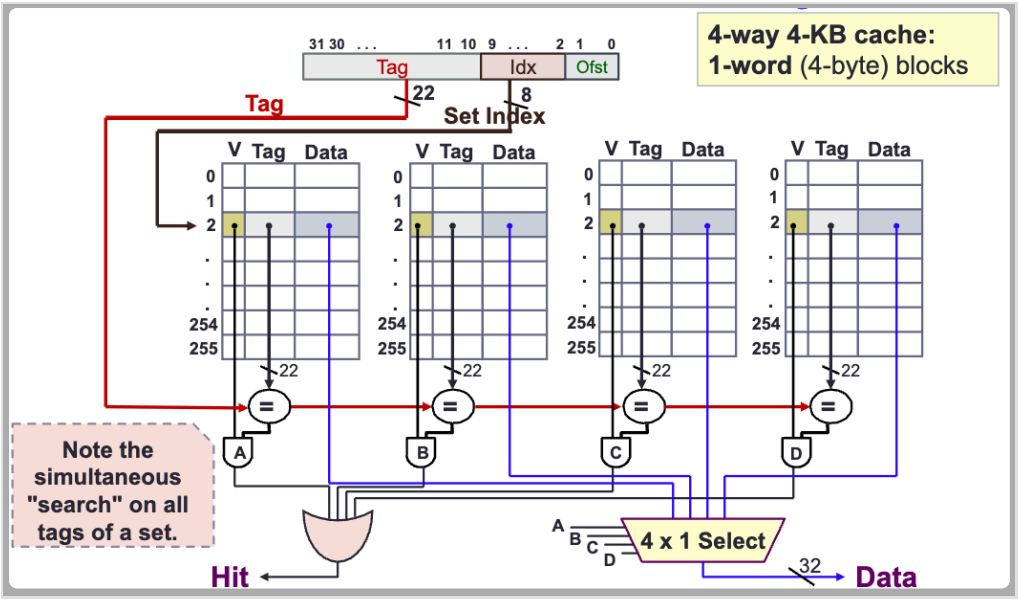
Core Concepts

Cache Set Index to allocate is determined from `Block Number % Number of Cache Sets`.

> ✓ Success
>
> It is similar to having multiple caches, we first try to put on the first one, and if not available can continue moving to the subsequent.



Cache Structure



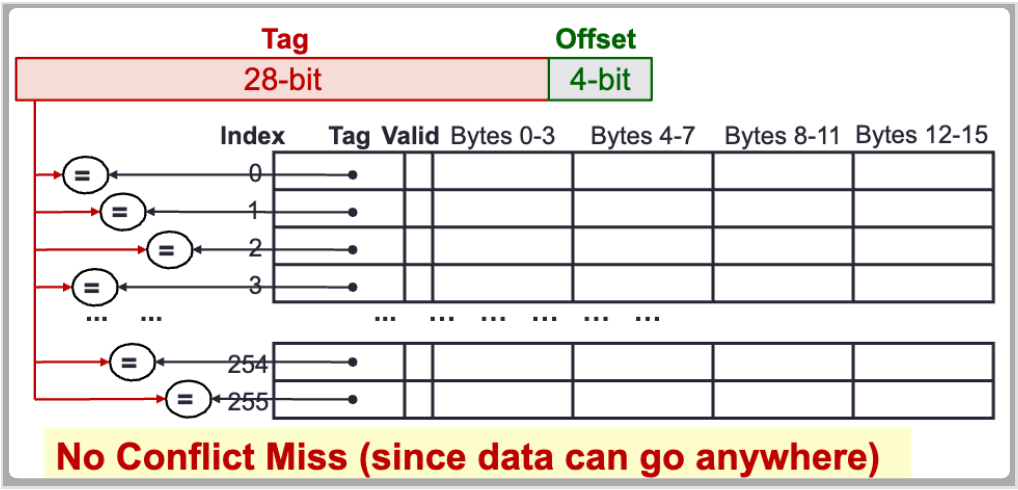
> **Rule of thumb**
>
> A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$.

Fully Associative Caching

Core Concepts

- A memory block can be placed in any location in the cache.
- There is no need for indexing. We can use entire block number as the tag.

Cache Structure



> **✓ Success**
>
> 1. Use a block replacement policy. For example, Least Recently Used (LRU).
> 2. When all cache blocks are occupied, rewrite the one selected from policy.
>
> Remark: We can also use this policy to select the block to replace in set associative.

The total miss is the sum of cold, conflict, and capacity miss. In this type of caching, there is no conflict miss.