# L02 Number Systems

- terminology

| Term | Definition |
| --- | --- |
| Bit | |
| Nibble | A group of **four** bits |
| Byte | A group of **eight** bits |
| Word | The chunks which microprocessors handle data in |
| Least significant bit | the bit in the 1's column |
| Most significant bit | the bit at the other end |
| Kilo | $2^{10}$ |
| Mega | $2^{20}$ |
| Giga | $2^{30}$ |

- 

| System | Base | N-digit Range | Usage |
| --- | --- | --- | --- |
| Decimal | 10 | $[0, 10^{N-1}]$ | |
| Binary | 2 | $[0, 2^{N-1}]$ | |
| Hexadecimal | 16 | $[0, 16^{N-1}]$ | |

- overflow - the result is **too big to fit** in the available digits
  - ex. 4-bit has the range $[0, 15]$, when the result exceeds 15, the fifth bit is **discarded**
- Sign/Magnitude representation
  - use the **msb** to indicate the sign **(0 + 1 -)**
  - the remaining $N - 1$ bits as the magnitude
  - ex. $5_{10} = 101_2$, $5_{10} = 0101_2$, $-5_{10} = 1101_2$
  - **ordinary binary addition** does **not** work for it

    - range: $[-2^{N-1}+1, 2^{N-1}-1]$
- two's complement representation
    - almost identical - except msb has a weight of $-2^{N-1}$
    - range: $[-2^{N-1}, 2^{N-1}-1]$
    - most positive: $01...111_2 = 2^{N-1}-1$, most negative: $10...000_2 = -2^{N-1}$
    - $-1 = 11...111_2$
    - steps: ex. find the representation of $-2_{10}$ as a $4$-bit two's complement number
        1. start with $+2_{10} = 0010_2$
        2. **invert** all the bits → $1101_2$
        3. add one → $1110_2$
    - ex. $5_{10} - 3_{10}$
        - $3_{10} = 0011_2$ → $-3_{10} = 1101_2$
        - $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$
- overflow detection
    - unsigned numbers - when the carry out of the most significant column is $1$
    - 2's complement numbers
        - two operands have the **same** sign bit, but the result has the **opposite** sign bit
        - **not** possible when adding a positive number to a negative number
- sign extension
    - ex. $3_{10} = 0011_2$ and $3_{10} = 1101_2$ → $3_{10} = 0000011_2$ and $3_{10} = 1111101_2$

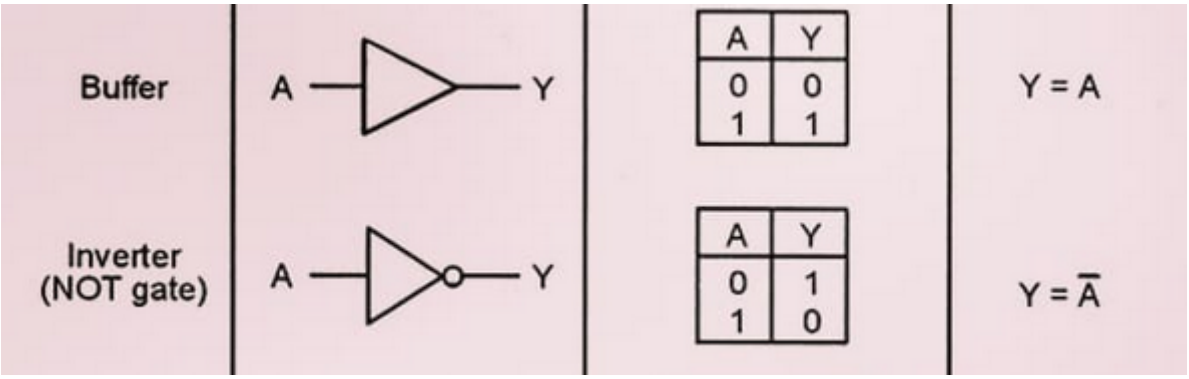
# L03

## Logic Gate

- logic gate - simple digital **circuits** that take one or more binary inputs and produce a binary output

**single-input logic gate**

- **NOT** gate - output is the **inverse** of its input
- **Buffer** - **copies** the input to the out put
  - similar to wires, but have some desirable characteristics
    - deliver large amounts of current to a motor
    - quickly send its output to many gates




- **triangle** symbol → buffer, **circle** → bubble (indicates inversion)

**two-input logic gate**

- **AND** gate - produces a TRUE output, iff **both** $A$ and $B$ are TRUE
  - $Y = A \cdot B, Y = AB, Y = A \cap B$
- **OR** gate - produces a TRUE output, if **either** $A$ **or** $B$ (or both) are TRUE
  - $T = A + B, Y = A \cup B$
- **XOR** gate - produces TRUE if $A$ or $B$, but **not both**, are TRUE
- **NAND** gate - NOT AND, produces TRUE **unless both** inputs are TRUE
- **NOR** gate - NOT OR, produces TRUE if **neither** $A$ **nor** $B$ is TRUE
- **XNOR** gate - produces TRUE if **both** inputs are FALSE or **both** inputs are TRUE

| AND | NAND | OR | NOR | XOR | XNOR |
|---|---|---|---|---|---|
| $AB$ | $\overline{AB}$ | $A+B$ | $\overline{A+B}$ | $A \oplus B$ | $\overline{A \oplus B}$ |
|  |  |  |  |  |  |

| B | A | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| B | A | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| B | A | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| B | A | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| B | A | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| B | A | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

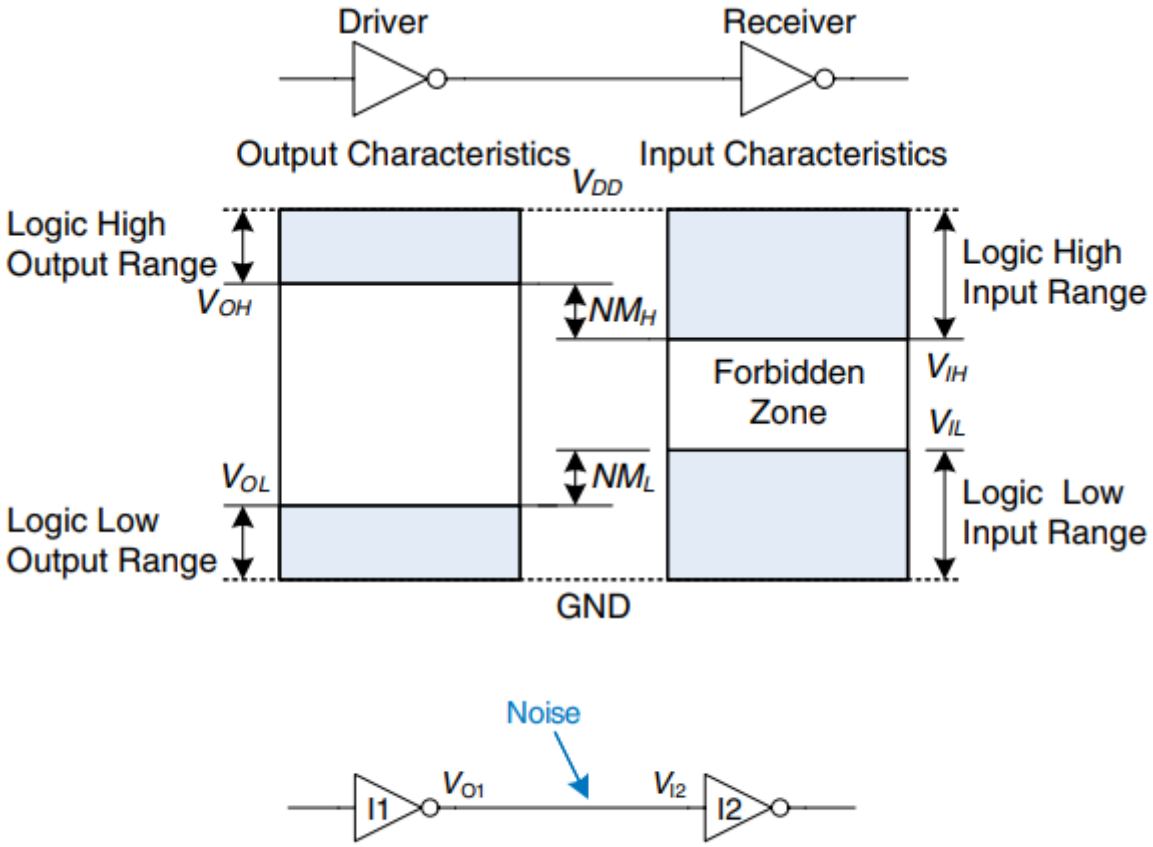
**multiple-input logic gate**

- **N-input XOR** gate - a parity gate and produces a TRUE output if an **odd** number of inputs are TRUE.

## Beneath the Digital Abstraction

- ground (GND) - $0\ V$, **lowest** voltage in the system
- $V_{DD}$ - comes from the power supply. **highest** voltage
  - initially, $5\ V$, now much lower
- logic levels - define the mapping of a **continuous** variable onto a discrete binary variable
  - defined to differentiate between $0$s and $1$s
  - first gate - driver, produces
    - a $LOW(0)$ output in range $0$ to $V_{OL}$, or
    - a $HIGH(1)$ output in range $V_{OH}$ to $V_DD$
  - second gate - receiver, get
    - range $0$ to $V_{IL}$ → consider input to be $LOW$
    - range $V_{IH}$ to $V_{DD}$ → consider input to be $HIGH$
    - for some reason (noise/faulty components/...), receiver's input should fall in the **forbidden zone** between $V_{IL}$ and $V_{IH}$, the behavior of the gate is unpredictable

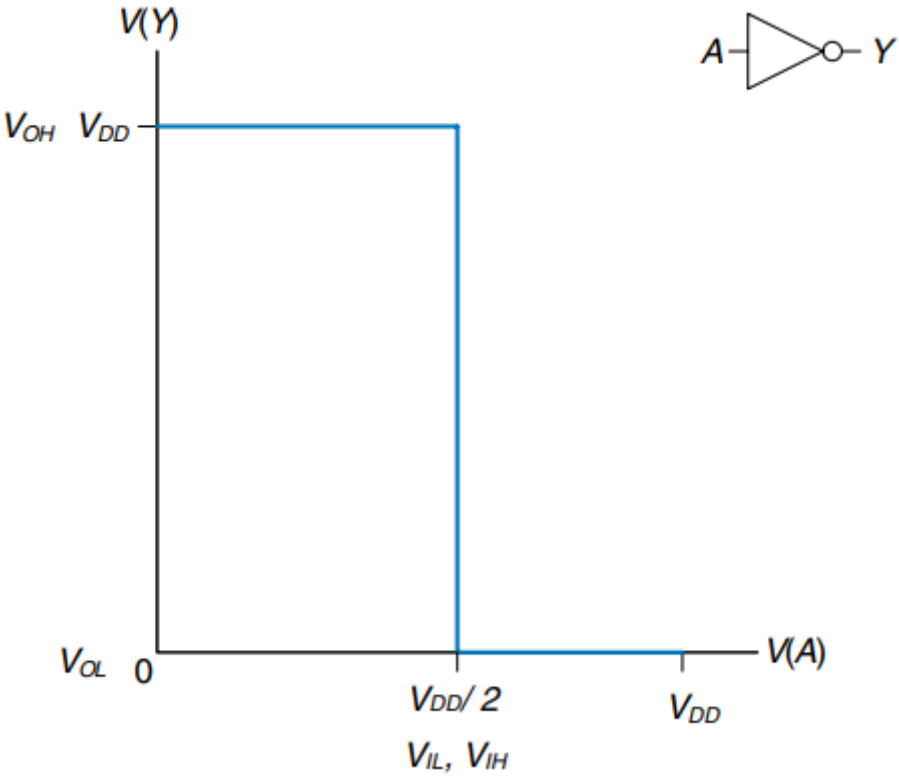
- correctly interpreted → must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$
  - even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level
  - noise margin - the amount of noise that could be added to a **worst-case** output such that the signal can still be interpreted as a valid input
    - $NM_L = V_{IL} - V_{OL}$
    - $NM_H = V_{OH} - V_{IH}$



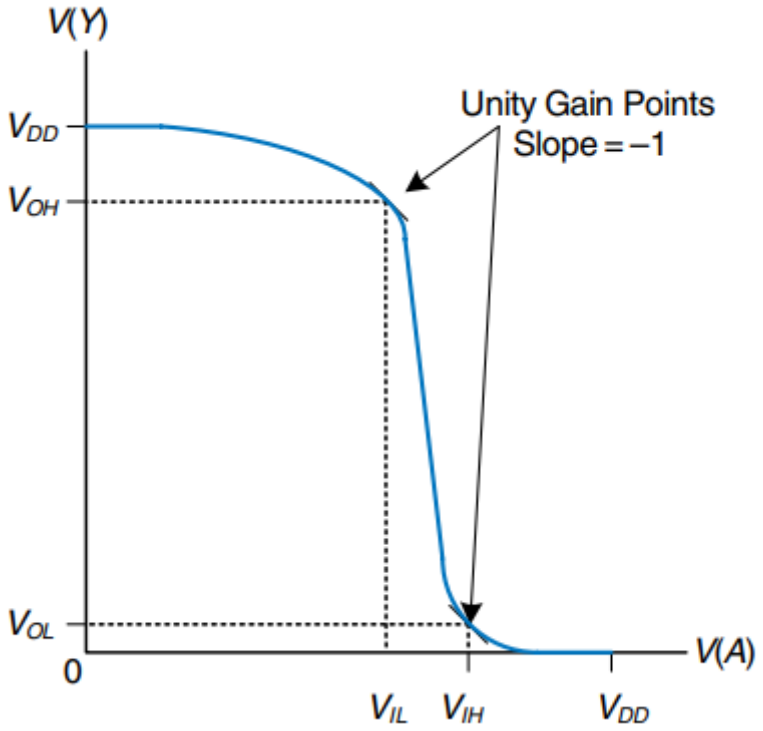

**DC transfer characteristics**

describe the output voltage as a function of the input voltage when the input is **changed slowly enough** that the output can keep up

- ideal inverter - have an abrupt switching threshold at $V_{DD}/2$
  - For $V(A) < V_{DD}/2, V(Y) = V_{DD}$
  - For $V(A) > V_{DD}/2, V(Y) = 0$
  - $V_{IH} = V_{IL} = V_{DD}/2$, $V_{OH} = V_{DD}$ and $V_{OL} = 0$




- real inverter - changes more gradually between extremes
  - when $V(A) = 0, V(Y) = V_{DD}$
  - when $V(A) = V_{DD}, V(Y) = 0$
  - the transition between these endpoints is smooth → how to define the logic levels
    - unity gain points - the slop of the transfer characteristic $dV(Y)/dV(A)$ is **-1**
    - choose logic levels at the unity gain points → **maximizes** the noise margins
      - $V_{IL}$ were reduced → $V_{OH}$ increase by a **small** amount
      - $V_{IL}$ were increased → $V_{OH}$ drop **precipitously**




**Static Discipline**

requires that, given logically valid inputs, every circuit element will produce logically valid output

- drawback - sacrifice the freedom of using arbitrary analog circuit elements
- benefits
  - simplicity and robustness of digital circuits
  - raise the level of abstraction from analog to digital
  - increasing design productivity by hiding needless detail

**logic family**

- all gates in a logic family obey the static discipline when used with other gates in the family
- logic gates in the same logic family snap together → use **consistent power supply voltages and logic levels**

| name | short term |
| --- | --- |
| Transistor-Transistor Logic | TTL |

| name | short term |
|---|---|
| Complementary Metal-Oxide-Semiconductor Logic | CMOS |
| Low Voltage TTL Logic | LVTTL |
| Log Voltage SMOS Logic | LVCMOS |

**Table 1.4 Logic levels of 5 V and 3.3 V logic families**

| Logic Family | $V_{DD}$ | $V_{IL}$ | $V_{IH}$ | $V_{OL}$ | $V_{OH}$ |
|---|---|---|---|---|---|
| TTL | 5 (4.75–5.25) | 0.8 | 2.0 | 0.4 | 2.4 |
| CMOS | 5 (4.5–6) | 1.35 | 3.15 | 0.33 | 3.84 |
| LVTTL | 3.3 (3–3.6) | 0.8 | 2.0 | 0.4 | 2.4 |
| LVCMOS | 3.3 (3–3.6) | 0.9 | 1.8 | 0.36 | 2.7 |

**Table 1.5 Compatibility of logic families**

| | | Receiver | | | |
|---|---|---|---|---|---|
| | | TTL | CMOS | LVTTL | LVCMOS |
| Driver | TTL | OK | NO: $V_{OH} < V_{IH}$ | MAYBE[a] | MAYBE[a] |
| | CMOS | OK | OK | MAYBE[a] | MAYBE[a] |
| | LVTTL | OK | NO: $V_{OH} < V_{IH}$ | OK | OK |
| | LVCMOS | OK | NO: $V_{OH} < V_{IH}$ | OK | OK |

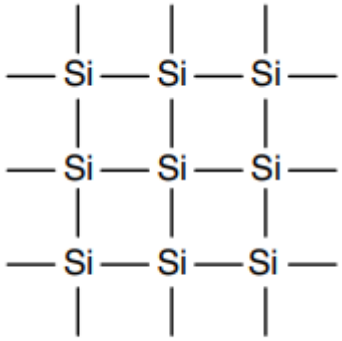
[a] As long as a 5 V HIGH level does not damage the receiver input

## CMOS Transistors

- transistor - electrically controlled switches that turn $ON$ or $OFF$ when a voltage or current is applied to a control terminal
  - bipolar transistors
  - metal-oxide-semiconductor field effect transistors (MOSFETs)

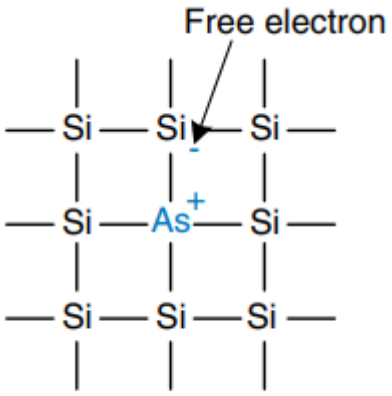
**Silicon**

- MOS transistors are built from silicon
- crystalline **lattice** → form a cubic crystal



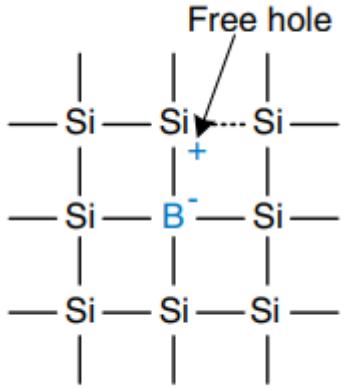
**(a)**

- a line represents a covalent bond
- by itself, silicon is a **poor** conductor
- become **better** conductor → add small amounts of **dopant** atoms (impurities)
- **n-type** dopant - group $V$ dopant added → extra electron that is not involved in the bonds → **free electron**



**(b)**

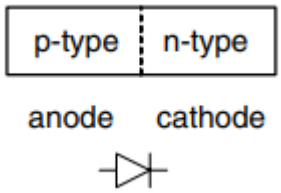
- **p-type** dopant - group III dopant added → missing electron → leaving a **hole** → **free hole**




- is a semiconductor since - the conductivity changes over many orders of magnitude depending on the concentration of dopants

**Diodes**

- diode - the junction between p-type and n-type silicon
  - **anode** - p-type region
  - **cathode** - n-type region

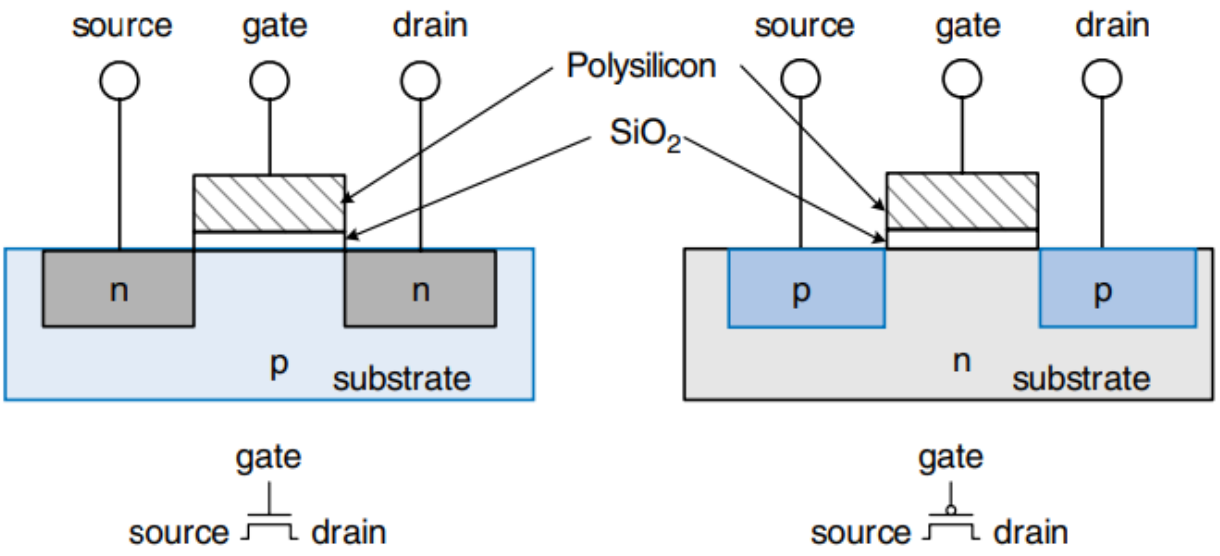



- diode is **forward biased** - voltage anode > voltage cathode
  - current flows from **anode** to the **cathode**
- diode is **reverse biased** - voltage anode < voltage cathode
  - **no** current flows

**nMOS and pMOS Transistors**

- several layers of conducting and insulating materials
  - conducting layer → **gate** (metal)
  - insulating layer → silicon dioxide → top of silicon wafer, **substrate**
- charge flows from the **source** to the **drain**

- nMOS - flow from **negative** to **positive**
- pMOS - flow from **positive** tot **negative**
- if draw schematics with the most + at the **top**
  - the source of (-) in nMOS is the **bottom**
  - the source of (+) in pMOS is the **top**

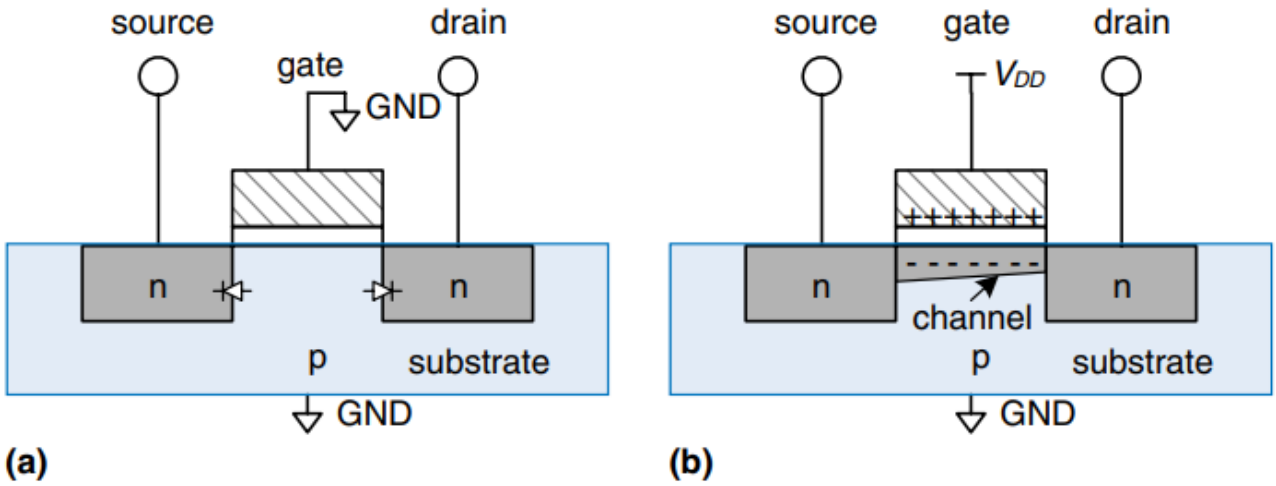



**(a)** nMOS **(b)** pMOS

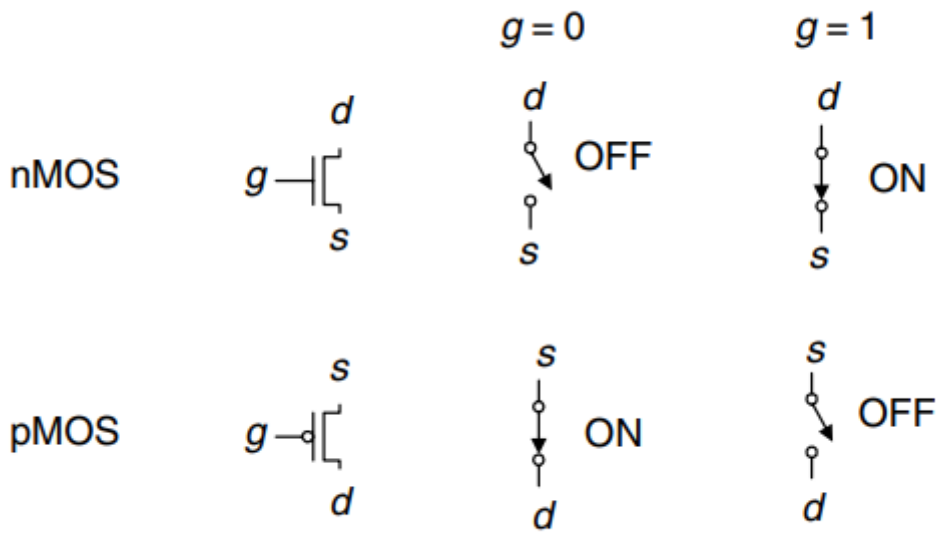
- nMOS
  - when gate is at $0V$ → reverse biased ← source or drain V is nonnega

    → **OFF**
  - when gate raised to $V_{DD}$ → top: +, bottom: - → large enough, inverts from p-type to n-type → inverted region is called the **channel**

    → **ON**
  - threshold voltage ($V_t$) - the gate voltage required to turn on a transistor, $0.3 - 0.7V$
  - pass $0$'s well but pass $1$'s poorly
    - when gate at $V_{DD}$ - drain only swing between $0$ and $V_{DD} - V_t$




- pMOS - work in the opposite fashion
  - when gate is at $0V$ → ON
  - when gate raised to $V_{DD}$ → OFF
  - pass $1$'s well but pass $0$'s poorly
- CMOS (Complementary MOS) - provide both flavors of transistors
  - start with a p-type wafer
  - implant n-type regions (wells) where the pMOS transistors should go



**CMOS Logic Gates**

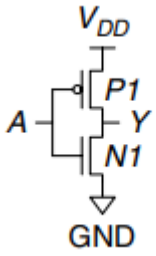
wires are always **joined** at three-way junctions

joined at four-way junctions **only if** a dot is shown

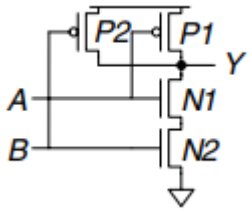
- NOT gates
  - if $A = 0$, $N1$ is OFF and $P1$ is ON

    → $Y$ connected to $V_{DD}$ → pulled up to a logic **1**
  - if $A = 1$, $N1$ is ON and $P1$ is OFF

    → $Y$ connected to $GND$ → pulled down to a logic $0$



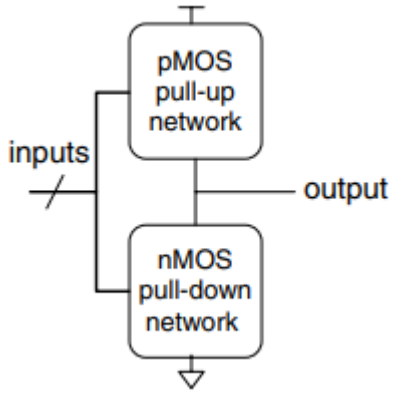

- NAND gates



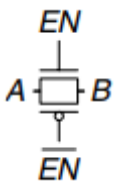

| A | B | Pull-Down Network | Pull-Up Network | Y |
|---|---|---|---|---|
| 0 | 0 | OFF | ON | 1 |
| 0 | 1 | OFF | ON | 1 |
| 1 | 0 | OFF | ON | 1 |
| 1 | 1 | ON | OFF | 0 |

- general form of an inverting logic gate
  - and → series
  - or → parallel

| Pull-Up | Pull-Down | Output |
|---|---|---|
| OFF | OFF | floating |
| OFF | ON | 0 (i.e., GND) |
| ON | OFF | 1 (i.e., $V_{DD}$) |
| ON | ON | *short circuit* |




- transmission/pass gates
  - parallel combination of pMOS and nMOS
  - pass both $1$'s and $0$'s well
  - **enables** - control signal, $EN$ and $\overline{EN}$
  - when $EN = 0$ and $\overline{EN} = 1$ → **both OFF**
  - when $EN = 1$ and $\overline{EN} = 0$ → **both ON**



# L04 Combinational Circuits

## Boolean Algebra

- terminology

| Term | Definition |
| --- | --- |
| complement (of a variable) | the inverse of the variable |
| literal | the variable or its complement |
| True vs. Complementary Form | $A$ vs. $\overline{A}$ |
| Product | the AND of one or more literals |
| Sum | the OR of one or more literals |
| minterm vs. maxterm | product (sum) involving all of the inputs to the function |
| Canonical forms | the sum of products |

- DeMorgan's Theorem
  - the complement of the product is the sum of the complements
  - the complement of the sum is the product of the complements

| A | B | C | minterm | minterm name | maxterm | maxterm name |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | $\bar{A} \cdot \bar{B} \cdot \bar{C}$ | $m_0$ | $A + B + C$ | $M_0$ |
| 0 | 0 | 1 | $\bar{A} \cdot \bar{B} \cdot C$ | $m_1$ | $A + B + \bar{C}$ | $M_1$ |
| 0 | 1 | 0 | $\bar{A} \cdot B \cdot \bar{C}$ | $m_2$ | $A + \bar{B} + C$ | $M_2$ |
| 0 | 1 | 1 | $\bar{A} \cdot B \cdot C$ | $m_3$ | $A + \bar{B} + \bar{C}$ | $M_3$ |
| 1 | 0 | 0 | $A \cdot \bar{B} \cdot \bar{C}$ | $m_4$ | $\bar{A} + B + C$ | $M_4$ |
| 1 | 0 | 1 | $A \cdot \bar{B} \cdot C$ | $m_5$ | $\bar{A} + B + \bar{C}$ | $M_5$ |
| 1 | 1 | 0 | $A \cdot B \cdot \bar{C}$ | $m_6$ | $\bar{A} + \bar{B} + C$ | $M_6$ |
| 1 | 1 | 1 | $A \cdot B \cdot C$ | $m_7$ | $\bar{A} + \bar{B} + \bar{C}$ | $M_7$ |

## Schema Guidelines

- inputs are on the left (or top) side of a schematic
- outputs are on the right (or bottom) side of a schematic
- whenever possible, gates should flow from left to right
- straight wires are better
- wires always connect at a $T$ junction
- a dot where the wires cross indicates a connection between the wires
- wires crossing **without** a dot make no connection

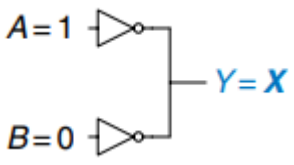
## Bubble Pushing

- begin at the output of the circuit and work toward the inputs
- push any bubbles on the final output back toward the input
- working backward, draw each gate in a form so that bubbles cancel
  - if the gate ... an input bubble → draw the preceding gate ... an output bubble
    - has → with
    - does not has → without

## Illegal and Floating values

#### illegal values

- the symbol $X$ indicates that the circuit node has an **unknown** or **illegal** value
- **contention** - commonly happens if it is being driven to both $0$ and $1$ at the same time
  - error, must be avoided
  - the actual voltage on a node with it is <u>often</u> in the forbidden zone
  - cause large amounts of power to flow → getting hot and possibly damaged

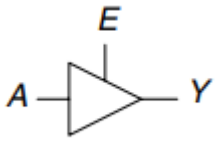



- $X$ values are sometimes used to indicate an **uninitialized** value

| $X$ appears | meaning |
|---|---|
| in truth table | the value of the variable is unimportant |
| in circuit | the circuit node has an unkown or illegal value |

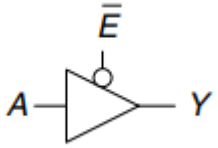
**floating value**

- the symbol $Z$ indicates that a node is being driven neither HIGH nor LOW
  - the node is said to be **floating, high impedance,** or high $Z$
- typical misconception - a floating or undriven node is the **same** as a logic $0$
- common way
  - forget to connect a voltage to a circuit input
  - assume that an unconnected input is the same as an input with the value of $0$
- tristate buffer (active high enable)
  - three possible output states: HIGH (1), LOW (0), floating (Z)
  - input A, output Y, enable E
  - when E is TRUE → same as buffer
  - when E is FALSE → output is allowed to float



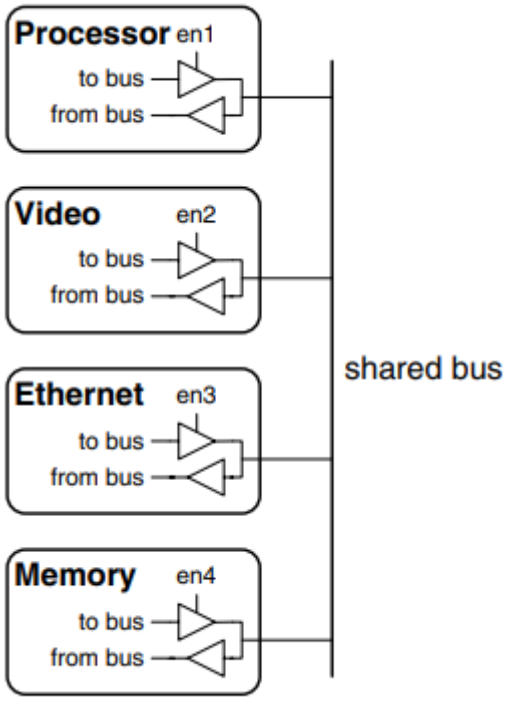

| E | A | Y |
|---|---|---|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

  - active low enable




| $\overline{E}$ | A | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | Z |
| 1 | 1 | Z |

- shared bus
  - connect multiple chips
  - only one chip at a time is allowed to assert its enable signal to drive a value
  - other chips must produce floating outputs




## Karnaugh Maps

- measuring cost
  - gate cost (G) - number of gates
  - gate cost with inverters (GN) → number of gates, including inverters

- we can use Boolean algebra to minimize equations in sum-of-products form

$$Y = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C = \overline{A}\overline{B}(\overline{C} + C) = \overline{A}\overline{B}$$

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(a)

| Y \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| C = 0 | 1 | 0 | 0 | 0 |
| C = 1 | 1 | 0 | 0 | 0 |

(b)

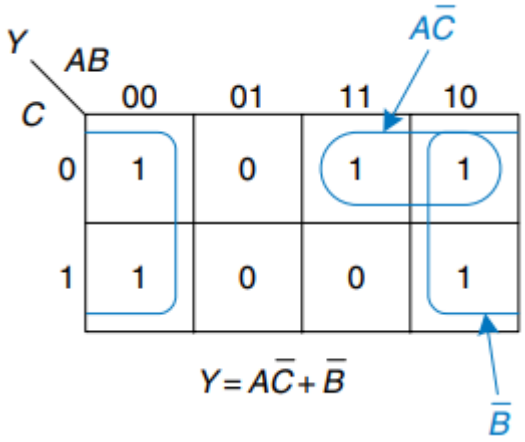
| Y \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| C = 0 | $\overline{A}\overline{B}\overline{C}$ | $\overline{A}B\overline{C}$ | $AB\overline{C}$ | $A\overline{B}\overline{C}$ |
| C = 1 | $\overline{A}\overline{B}C$ | $\overline{A}BC$ | $ABC$ | $A\overline{B}C$ |

(c)

**Figure 2.43** **Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms**

| Y \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| C = 0 | 1 | 0 | 0 | 0 |
| C = 1 | 1 | 0 | 0 | 0 |

- **prime implicants** - an implicant that cannot be combined with another to form a new implicant with fewer literals
  - implicants - the product of one or more literals
- rules
  - use the fewest circles necessary to cover all the $1$'s
  - all the squares in each circle must contain $1$'s
  - each circle must span a block that is a power of $2$
  - each circle should be as large as possible
  - a circle may wrap around the edge
  - a $1$ in a K-Map may be circled multi times




- don't care $X$

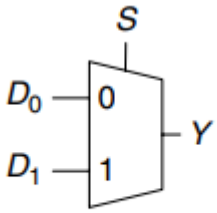
# L05 Optimization and Mapping

## Multiplexers

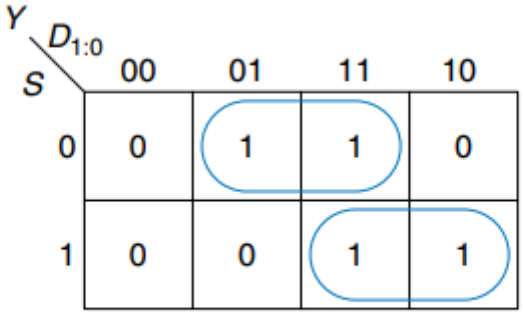
- are among the most commonly used combinational circuits
- choose an output from among several possible inputs based on the value of a **select** signal
- sometimes affectionately called a **mux**
- inputs: M select, N=$2^M$ data, output: 1
- using $M$ bits, select exactly one of $N$ data inputs to be the output

#### 2:1 multiplexer

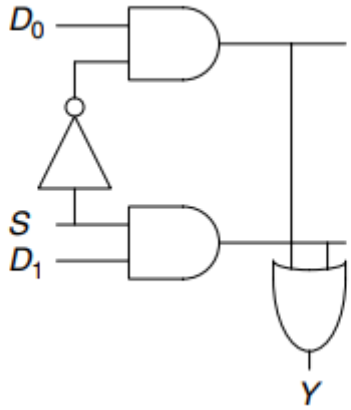
- two data inputs $D_0$ and $D_1$, a select input $S$, one output $Y$
- chooses between the two data inputs based on the select
  - $S = 0 \rightarrow Y = D_0$
  - $S = 1 \rightarrow Y = D_1$
- $S$ also called a control signal



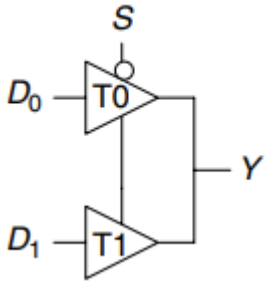

| S | $D_1$ | $D_0$ | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




$$Y = D_0\bar{S} + D_1S$$

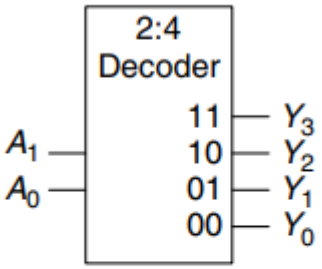



- can be built from tristate buffers
  - when $S = 0$, tristate T0 is enabled → $D_0$ flow to Y
  - when $S = 1$, tristate T1 is enabled → $D_1$ flow to Y




## Decoders

- inputs: M select, 1 data, output: $N = 2^M$
- using $M$ bits, forward the input to exactly one of $N$ outputs
- logic functions can be implemented with a decodes and an OR gate




| $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

## Look Up Tables

- consists of an $N:1$ multiplexer where:
  - each of its $N$ inputs are connected to a $1$-bit memory cell
  - the select input chooses which $1$-bit memory cell to "look up"

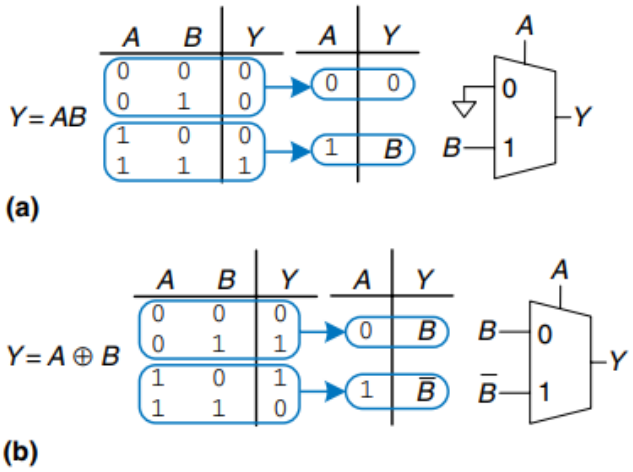


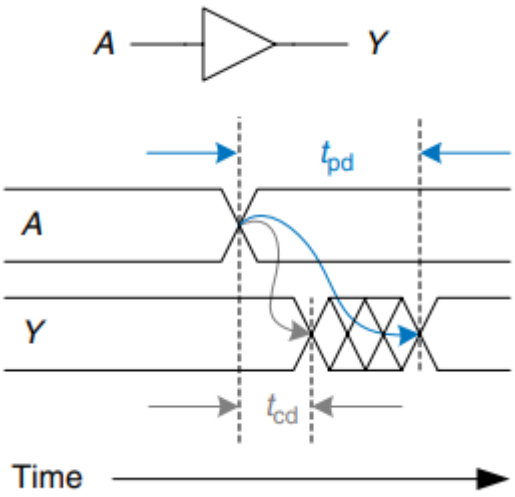

**Figure 2.60** Multiplexer logic using variable inputs

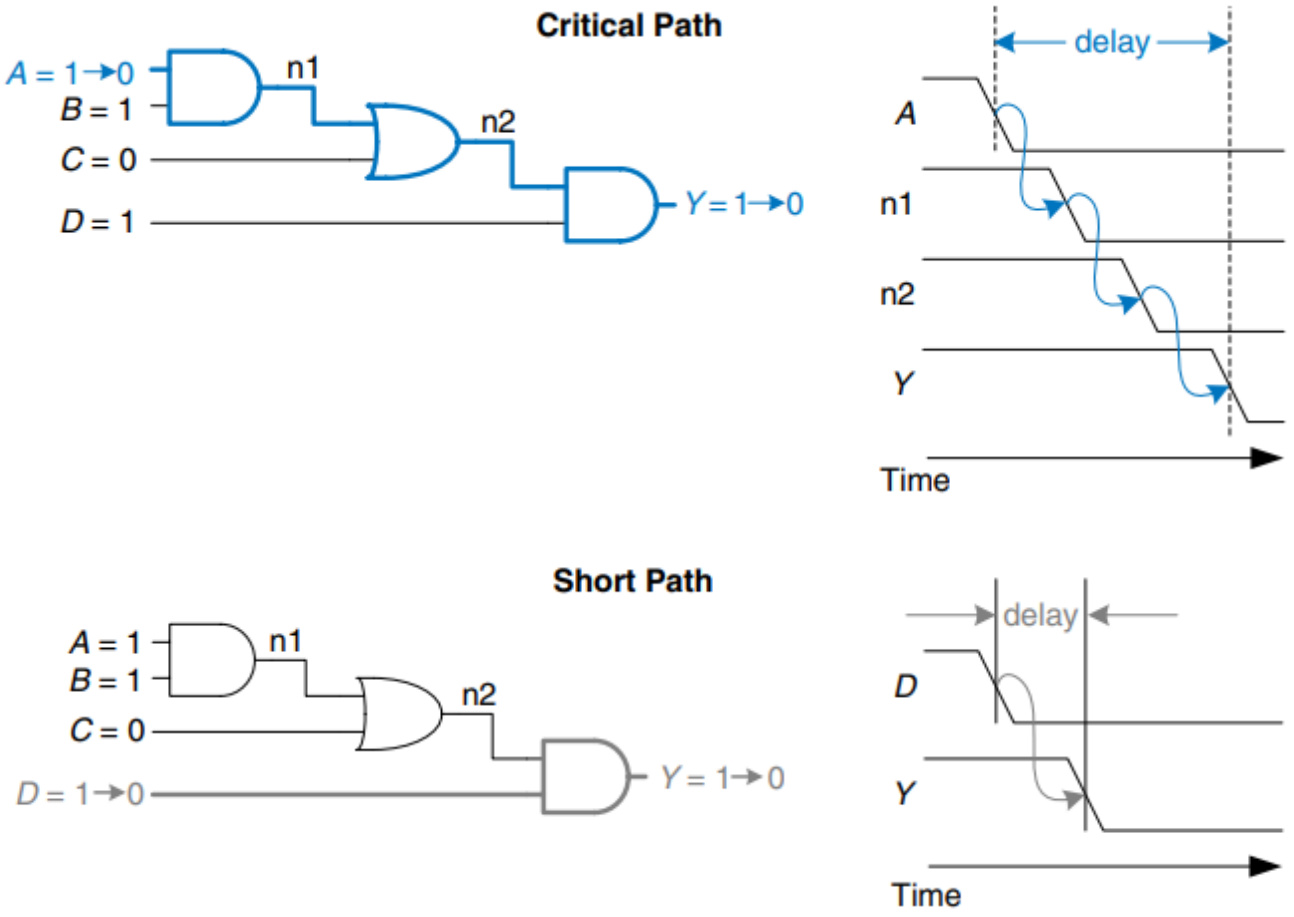
# L06 Arithmetic Circuits

## Timing

- terminology

| Term | Definition |
| --- | --- |
| Rising edge | the transition from LOW to HIGH |
| Falling edge | the transition from HIGH to LOW |
| Propagation delay | $t_{pd}$, the maximum time from when an input changes until the output or outputs reach their final value |
| Contamination delay | $t_{cd}$, the minimum time from when an input changes until any output starts to change its value |
| Path | a signal takes from input to output |
| Critical path | the longest and the slowest path |
| Short path | the shortest and the fastest path |




- calculation
  - the $t_{pd}$ of a combinational circuit is the sum of $t_{pd}$ through each element on the **critical path**
  - the $t_{cd}$ is the sum of the $t_{cd}$ through each element on the short path

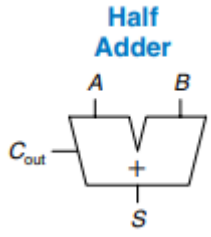
**Table 2.7 Timing specifications for multiplexer circuit elements**

| Gate | $t_{pd}$ (ps) |
|---|---|
| NOT | 30 |
| 2-input AND | 60 |
| 3-input AND | 80 |
| 4-input OR | 90 |
| tristate ($A$ to $Y$) | 50 |
| tristate (enable to $Y$) | 35 |

## Addition and Subtracting

### Half Adder (1-bit)

- $S$ is the sum of $A$ and $B$
- $C_{\text{out}}$ is the carry out

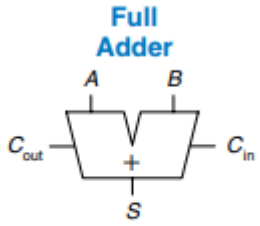



| A | B | $C_{out}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$S = A \oplus B$$
$$C_{out} = AB$$

### Full Adder (1-bit)

- $C_{\text{in}}$ is to accept $C_{\text{out}}$ of the previous column

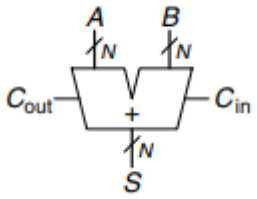



| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

**Garry Propagate Adder ($N$-bits)**

- $A$ and $B$ - $N$-bit inputs
- commonly called a **carry propagate adder** (CPA)
  - the carry out of one bit propagates into the next bit
- three common CPA implementations
  - ripple-carry adders
  - carry-lookahead adders
  - prefix adders

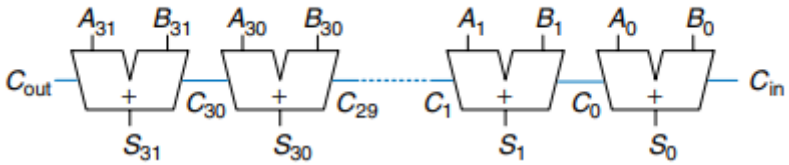



**Ripple-carry adders**

- chain together $N$ full adders
- the $C_{\text{out}}$ of one stage acts as the $C_{\text{in}}$ of the next stage
- benefit
  - reused the full adder module many times to form a large system

- drawback
  - being slow when $N$ is large → forth all the way back to $C_{\text{in}}$
- the carry *ripples* through the carry chain
- the delay of the adder, $t_{\text{ripple}}$
  - **grows directly** with the number of bits
  - $t_{\text{FA}}$ is the delay of a full adder

$$t_{\text{ripple}} = Nt_{FA}$$




**Garry-Lookahead Adder**

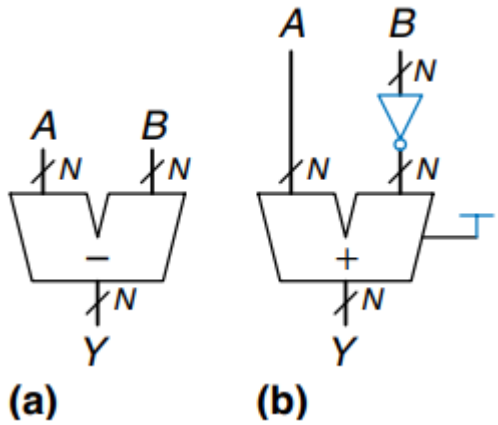
- dividing the adder into blocks and providing circuitry
  - to quickly determine the carry out of a block as soon as the carry in is known
  - look ahead across the blocks rather than waiting to ripple through all the full adders inside a block
- use **generate** (G) and **propagate** (P) signals → how a column or block determines the carry out
- the $i$th column of an adder is said to generate a carry
  - if it produces a carry out **independent** of the carry in
  - guarantee to generate a carry, $C_i$, if $A_i$ and $B_i$ are both $1$
  - $G_i = A_iB_i$
- the column is said to propagate a carry
  - if it produces a carry out **whenever** there is a carry in
  - propagate a carry in $C_{i-1}$ if either $A_i$ or $B_i$ is $1$
  - $P_i = A_i + B_i$
- the $i$th column of an adder will generate a carry put, $C_i$, if it either generates a carry, $G_i$, or a propagates to carry in, $P_iC_{i-1}$

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1}$$

| Name | Delay | Area | Notes |
|---|---|---|---|
| Ripple-carry | Slow | Low | Re-uses full adders |
| Carry-lookahead | Fast | More | Generally faster when N > 16, but still linear |
| Prefix | Faster | More | Tree-like structure |

**Subtraction**

- flip the sign of the second number, then add
- $Y = A + \overline{B} + 1 = A - B$




## Shifters and Rotators

- shifter and rotators move bits and multiply or divide by powers of $2$
- logical shifter
  - shifts the number to the left (LSL) or right (LSR) and fills empty spots with $0$'s
  - 11001 LSL 2 = 001**00** ;$11001$ LSR 2 = **00**110
- arithmetic shifter
  - on **right** shifts fills the most significant bits with a **copy** of the old most significant bit(msb)
  - 11001 ASL 2 = 00100; 11001 ASR 2 = **11**110
- multiplication - left shift(LSL, ASL) by $N$ bits multiplies the number by $2^N$

- division - ASR by $N$ bits divides the number by $2^N$
- rotator - rotates number in circle such that empty spots are filled with bits shifted off the other end
  - 110**01** ROR 2 = **01**110; **11**001 ROL 2 = 001**11**

## Comparing

- comparators - determines whether two binary numbers are equal or if one is greater or less than the other
  - receives two $N$-bit binary number
  - **equality comparator** - produces a single output indicating whether $A$ equal to $B$
  - **magnitude comparator** - produces one or more outputs indicating the relative value of $A$ and $B$
    - subtract $B$ from $A$ → look at the result's msb
      - negative → $A < B$
      - otherwise → $A >= B$
- Arithmetic/Logical Unit (ALU) - combines a variety of mathematical and logical operations into a single unit

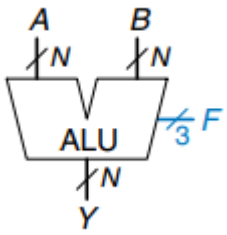


**Table 5.1 ALU operations**

| $F_{2:0}$ | Function |
|---|---|
| 000 | A AND B |
| 001 | A OR B |
| 010 | A + B |
| 011 | not used |
| 100 | A AND $\overline{B}$ |
| 101 | A OR $\overline{B}$ |
| 110 | A − B |
| 111 | SLT |

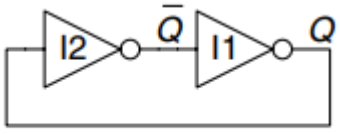
# L07 Sequential Circuits

## Bistatble element

- **bistable** circuit - the circuit element that has two stable states
  - why two stable states → binary digit $0$ and $1$

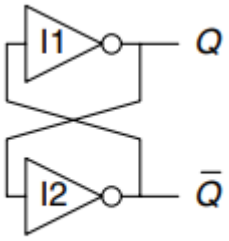
### inverter

- simple a pair of inverters connected in a loop



(a)

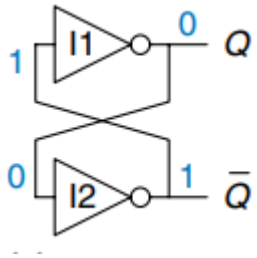
- **cross-coupled** inverters
  - the input of I1 is the output of I2 and vice versa



(b)

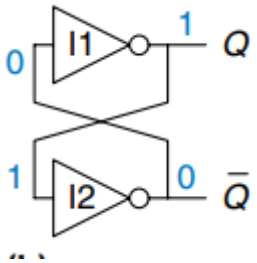
- case 1: $Q = 0$
  - I2 receives a FALSE input, $Q$, so it produces a TRUE output on $\overline{Q}$
  - I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on $Q$
  - this is consistent with the original assumption that $Q = 0$

    → stable




- case 2: $Q = 1$
  - I2 receives a TURE input, $Q$, so it produces a FALSE output on $\overline{Q}$
  - I1 receives a FALSE input, $\overline{Q}$, so it produces a TRUE output on $Q$
  - this is consistent with the original assumption that $Q = 1$

    → stable

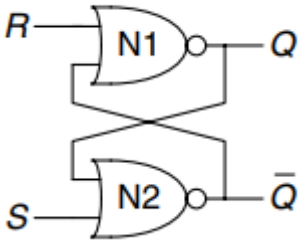



→ bistable

- the circuit has a third possible state with both outputs approximately halfway between 0 and 1 → **metastable** state
- not practical → the user has no inputs to **control** the state

**SR Latch**

- compose of two cross-coupled **NOR** gates
  - $S$ and $R$ inputs, which set and reset the output $Q$




| Case | $S$ | $R$ | $Q$ | $\bar{Q}$ |
|---|---|---|---|---|
| IV | 0 | 0 | $Q_{prev}$ | $\bar{Q}_{prev}$ |
| I | 0 | 1 | 0 | 1 |
| II | 1 | 0 | 1 | 0 |
| III | 1 | 1 | 0 | 0 |

- positives
  - can control state
    - can set $Q$ to $1$ by asserting the input $S$
    - can reset $Q$ to $0$ by asserting the input $R$
  - can **retain** state
    - input $S$ and $R$ are $0$
- negatives
  - $S = R = 1$ is awkward
  - asserting $S$ or $R$ dictates both:
    - what the state should be
    - when it should change

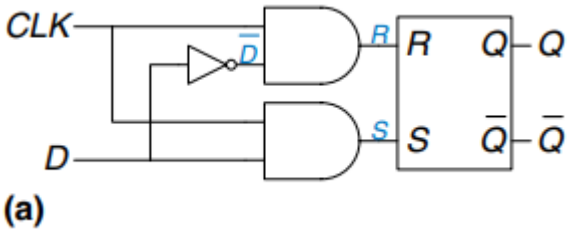
# Clock Element

### clock signals

- act as the "pulse" of a digital system
- change from $0$ to $1$ then 1 to 0 repeatedly
  - rising edge: a transition from 0 to 1
- period - time between rising edges of the clock
  - its inverse is frequency - number of rising edges per second

### D Latch

- data input $D$ - controls what the next state should be
- clock input, $CLK$ - controls when the state should change



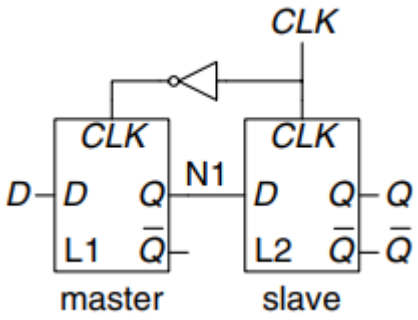

| CLK | D | $\overline{D}$ | S | R | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | X | $\overline{X}$ | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

(b)

- positive
  - can control state
    - CLK =1 → latch is transparent
  - can retain state
    - CLK = 0 → latch is opaque
- negative - always updates while CLK = 1

### D Flip-Flop

- built from two back-to-back D latches controlled by complementary clocks



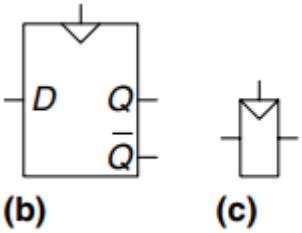
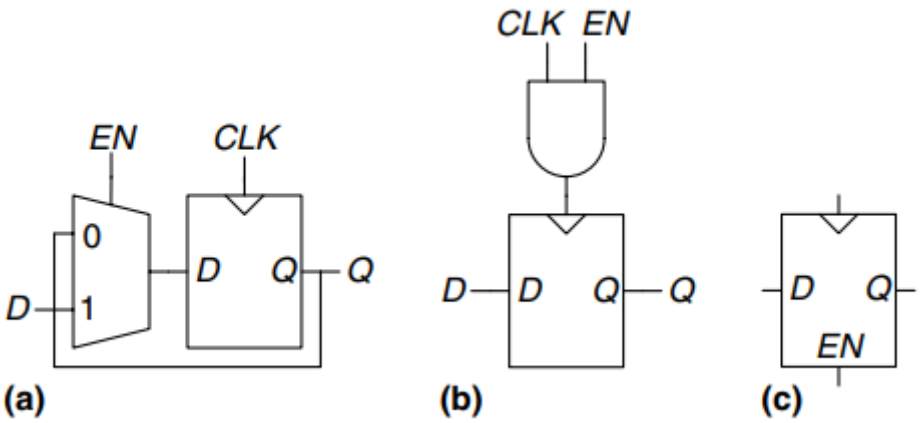



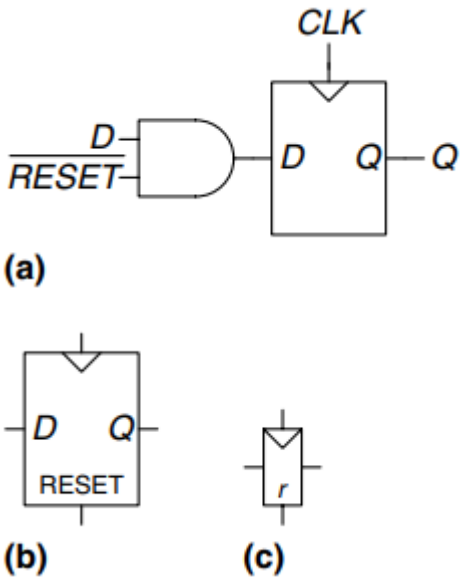

Table of truth:

| clk | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Q | $\overline{Q}$ |
| 0 | 1 | Q | $\overline{Q}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

- ! A D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other time

**Enabled Flip-Flop and Resettable Flip-Flop**






### The Enable Input

- EN = 1, D flip-flop behaves normally
- EN = 0, D flip-flop ignores CLK and retains state

Useful when you want to load a new value only some of the time.

### The Reset Input

- RESET = 0, D flip-flop behaves normally
- RESET = 1, D flip-flop ignores D and resets the output to 0

Useful when you "turn on" a system and want everything to start with 0.

**Register**

- an $N$-bit register is a bank of $N$ flip-flops that share a common CLK input, so that all bits of the register are updated at the same time

# L08 Finite State Machines

- a circuit with $k$ registers can be in one of $2^k$ states
- has M inputs, N outputs, and $k$ bits of state
- consists of two blocks of combinational logic → **next state logic** and **output logic**, and a register that stores that state
- next sate logic
  - computed based on the current state and inputs

# Synchronous logic design

- terminology

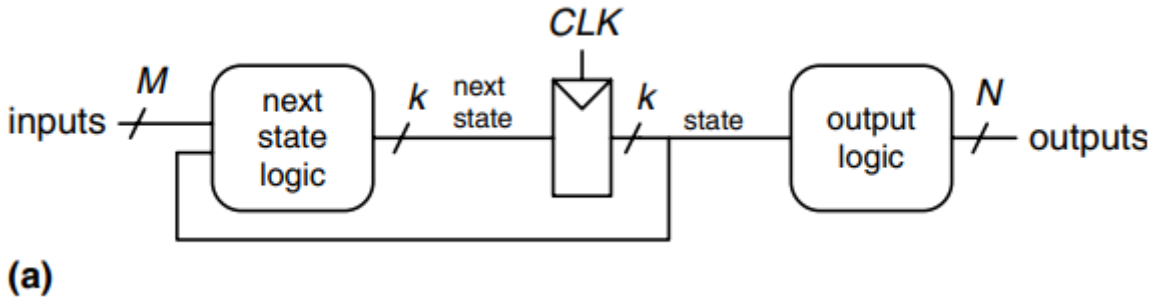
| Term | Definition/ Description |
|---|---|
| Synchronous | outputs are directly def back to input |
| Astable | the circuit has no stable states |
| Race condition | the behavior of the circuit depends on which of two paths through logic gates is fastest |
| Asynchronous | a circuits having race condition |
| Synchronous sequential circuit | has a clock input, whose rising edges indicate a sequence of times at which state transitions occur |

- astable circuits
  - each node oscillates between $0$ and $1$ with a period → **ring oscillator**
    - is a sequential circuit with zero inputs and one output that changes periodically
  - the period depends on the propagation delay of each component
- we can systematically design circuits by specifying how a circuit transitions between each of its states

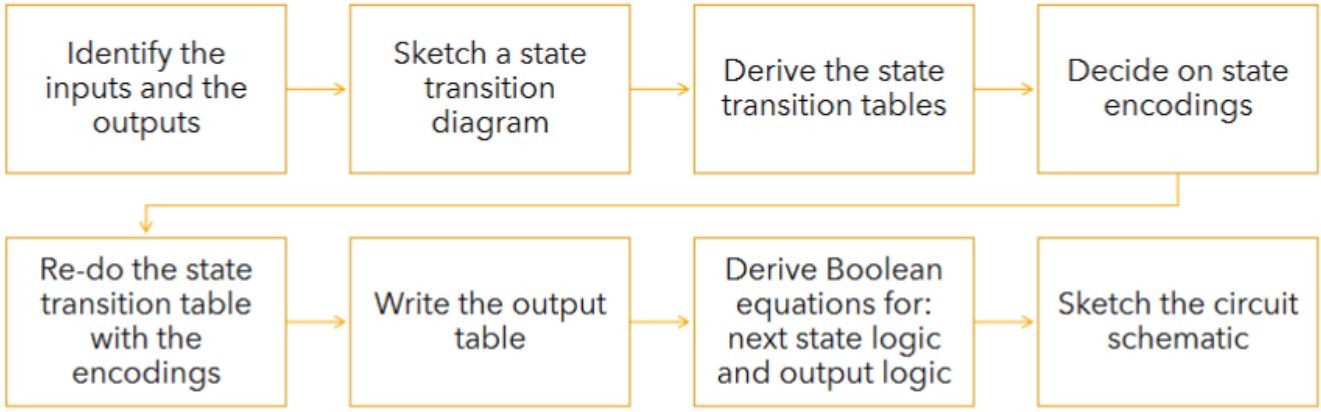
> - Every circuit element is either a register or a combinational circuit
> - At least one circuit element is a register
> - All registers receive the same clock signal
> - Every cyclic path contains at least one register.

# Moore FSM

### Design Step

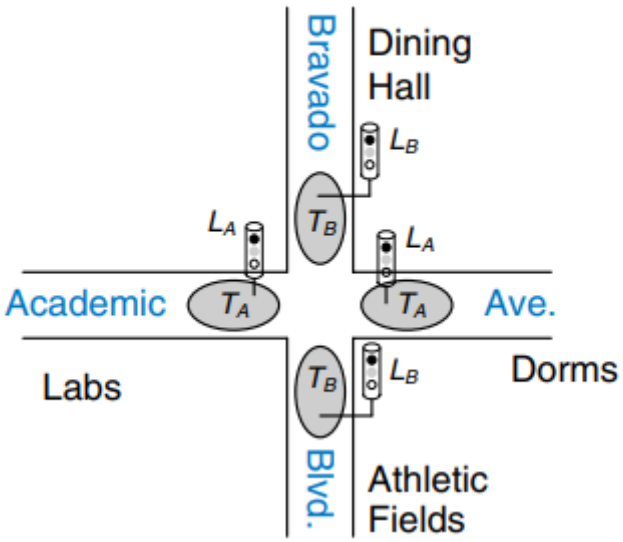


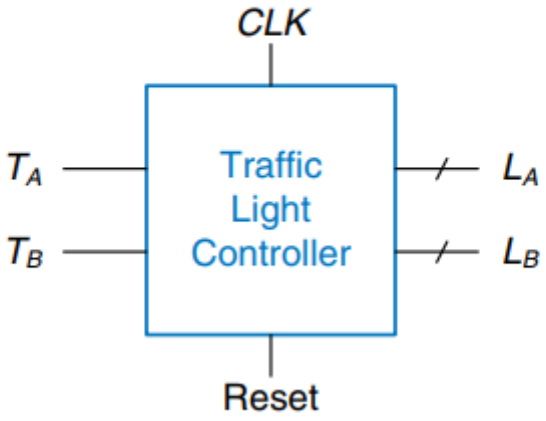

(a)



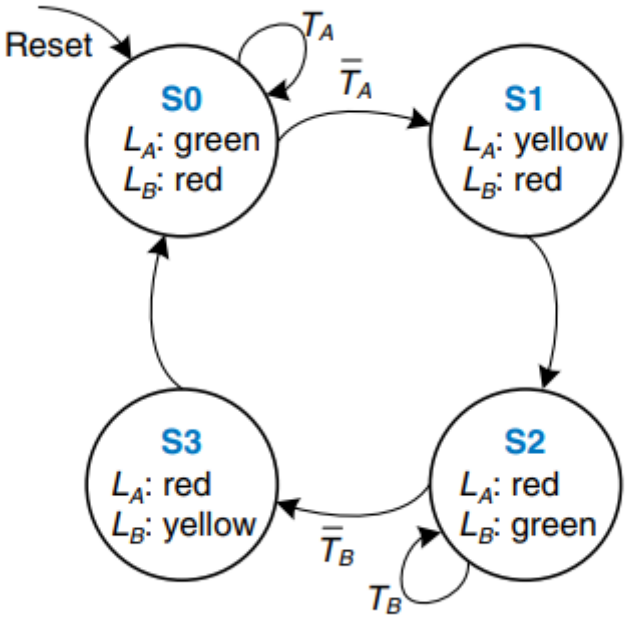

**Inputs and outputs**

- inputs - traffic sensors $T_A, T_B$
  - TRUE - students are present
  - FALSE - students are not present
- outputs - traffic lights $L_A, L_B$
  - red, green, blue




**State transition diagram**




- we do not bother to show the clock on the diagram
- since it is always present in a synchronous sequential circuit
- the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur

**State transition table**

| Current State $S$ | Inputs $T_A$ | Inputs $T_B$ | Next State $S'$ |
|---|---|---|---|
| S0 | 0 | X | S1 |
| S0 | 1 | X | S0 |
| S1 | X | X | S2 |
| S2 | X | 0 | S3 |
| S2 | X | 1 | S2 |
| S3 | X | X | S0 |

- indicates, for each state and input, what the next state should be
- uses don't care symbols (X) whenever the next state does not depend on a particular input

- Reset is omitted from the table
  - use resettable flip-flops that always go to state $S0$ on reset, independent of the inputs

**Binary encoding**

**Table 3.2 State encoding**

| State | Encoding $S_{1:0}$ |
|---|---|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

**Table 3.3 Output encoding**

| Output | Encoding $L_{1:0}$ |
|---|---|
| green | 00 |
| yellow | 01 |
| red | 10 |

$$S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B$$

$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

| Current State | | Inputs | | Next State | |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $T_A$ | $T_B$ | $S'_1$ | $S'_0$ |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

**Table 3.5 Output table**

| Current State | | Outputs | | | |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $L_{A1}$ | $L_{A0}$ | $L_{B1}$ | $L_{B0}$ |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

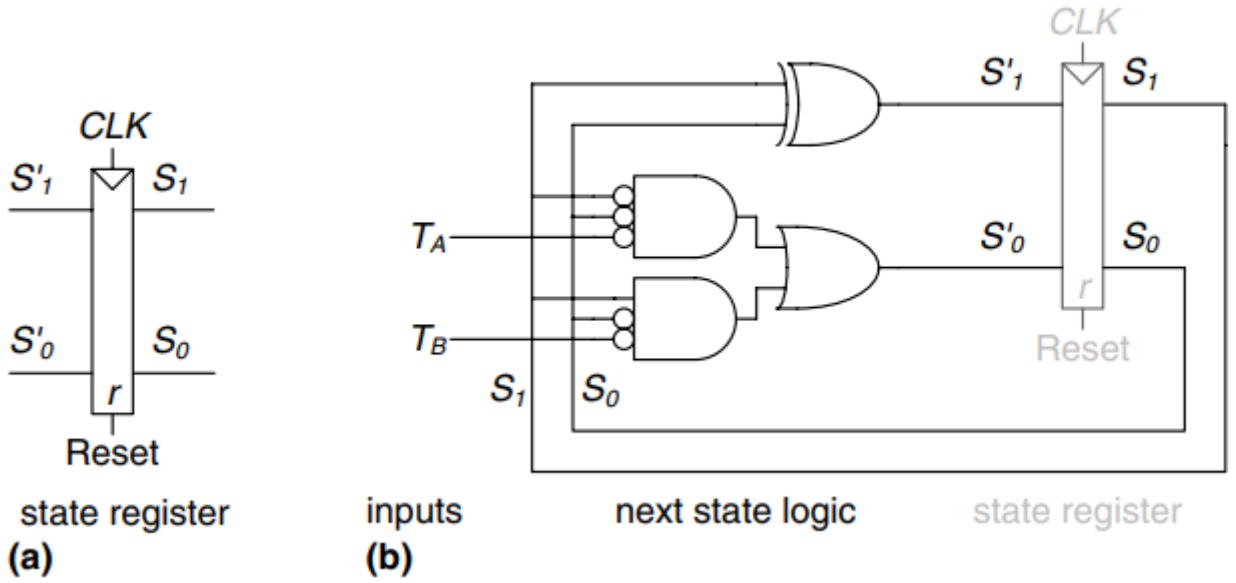
$$L_{A1} = S_1$$

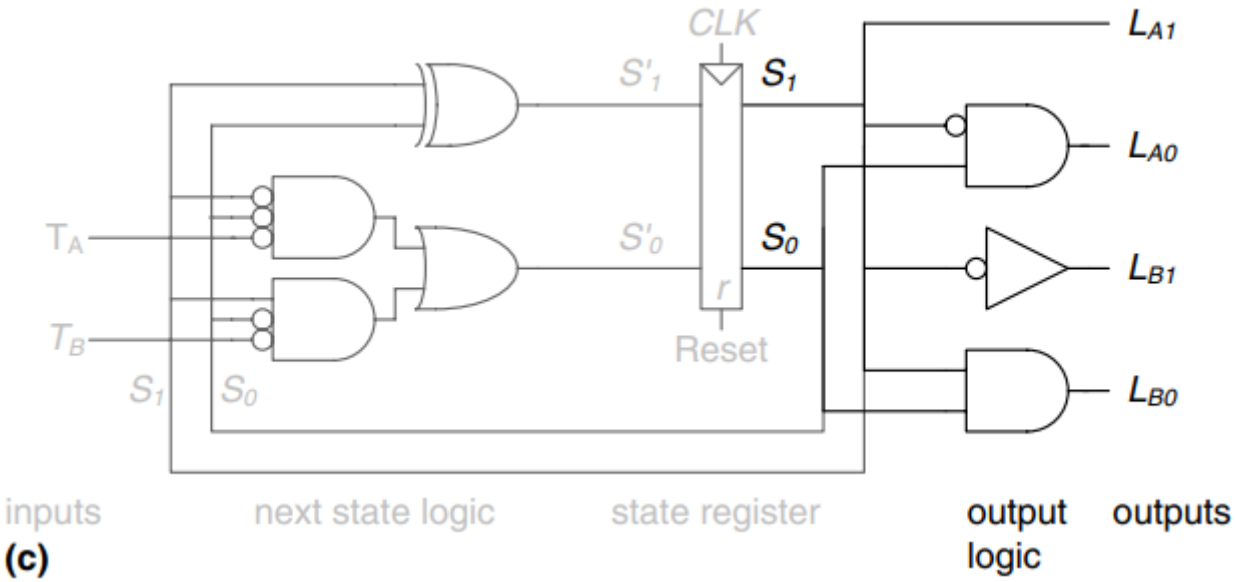
$$L_{A0} = \overline{S}_1 S_0$$

$$L_{B1} = \overline{S}_1$$

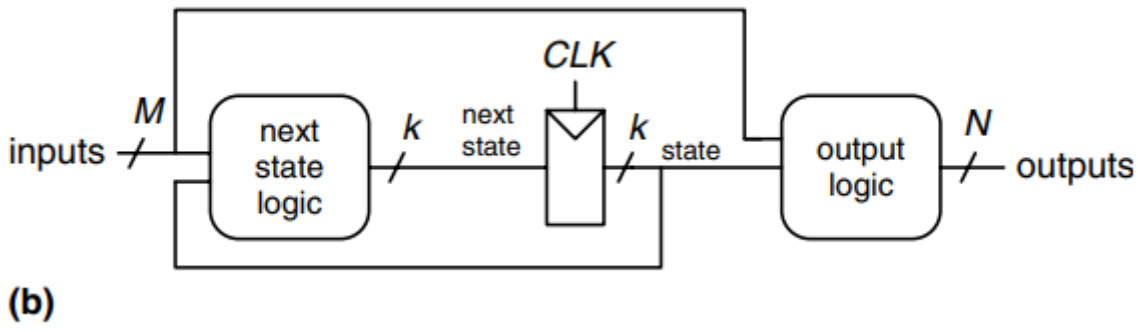
$$L_{B0} = S_1 S_0$$

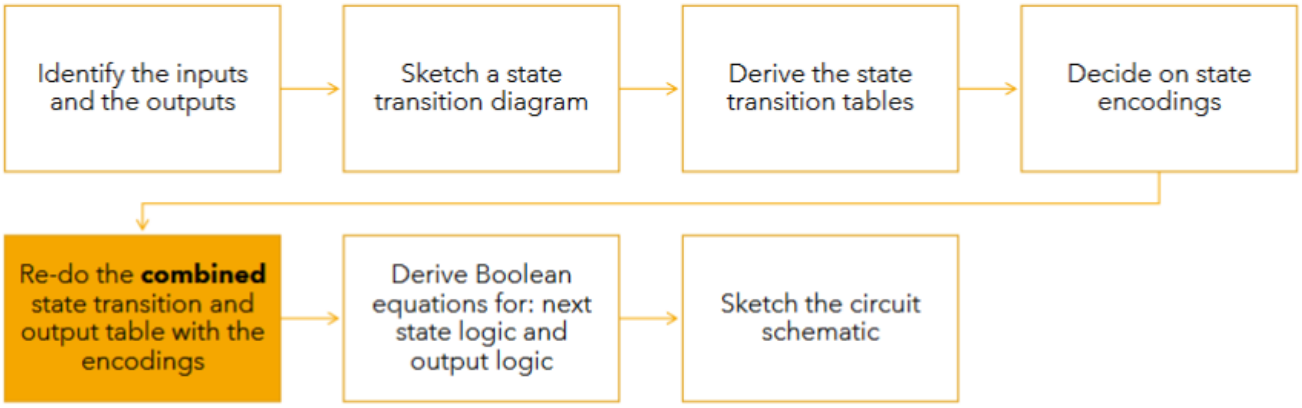
**schematic**

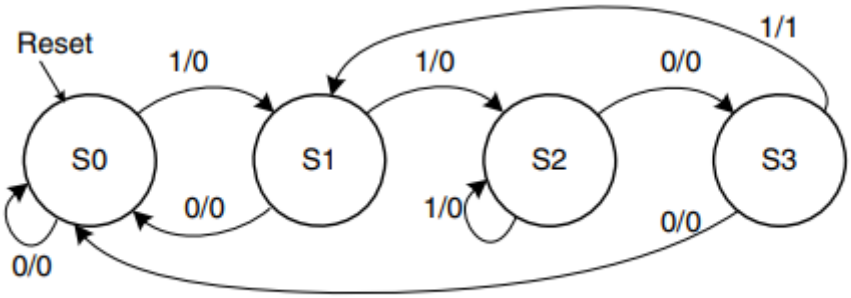



## Mealy FSM






(b)

**Table 3.15 Mealy state transition and output table**

| Current State $S$ | Input $A$ | Next State $S'$ | Output $Y$ |
|---|---|---|---|
| S0 | 0 | S0 | 0 |
| S0 | 1 | S1 | 0 |
| S1 | 0 | S0 | 0 |
| S1 | 1 | S2 | 0 |
| S2 | 0 | S3 | 0 |
| S2 | 1 | S2 | 0 |
| S3 | 0 | S0 | 0 |
| S3 | 1 | S1 | 1 |

**Table 3.16 Mealy state transition and output table with state encodings**

| Current State $S_1$ | Current State $S_0$ | Input $A$ | Next State $S'_1$ | Next State $S'_0$ | Output $Y$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

# L09 System Timing

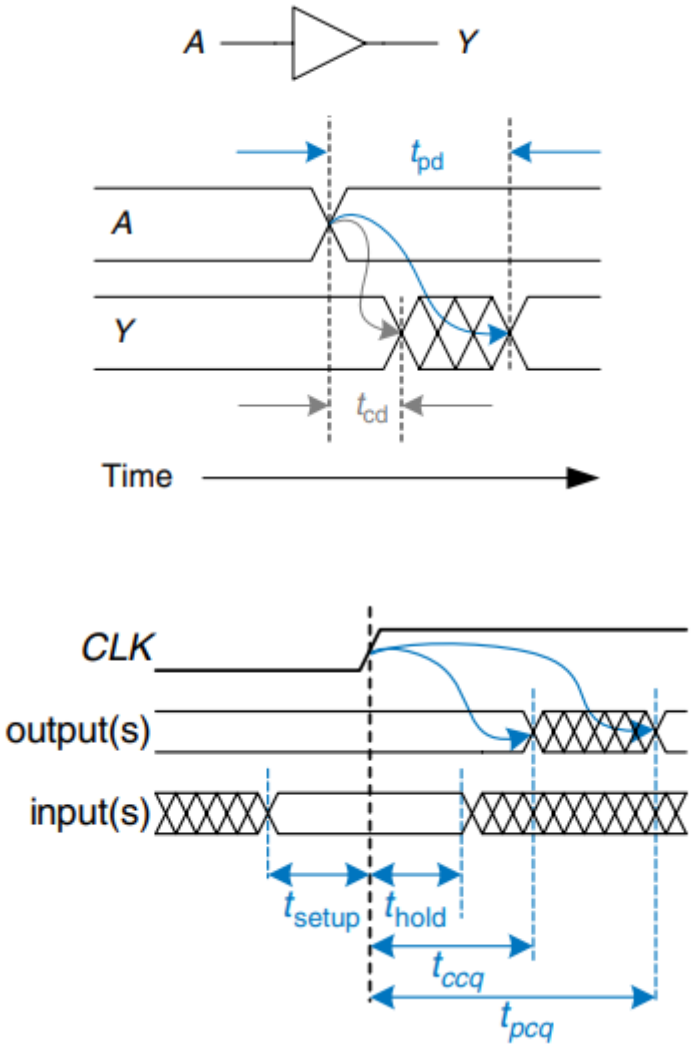
## Rules of Synchronous Sequential Circuits:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

- terminology
  - clock period - $T_c$ (cycle time)
  - clock frequency - $f_c = \frac{1}{T_c}$

| Term | Name | Notes |
|---|---|---|
| $t_{pd}$ | Propagation delay of combinational logic | depends on critical path |
| $t_{cd}$ | Contamination delay of combinational logic | depends on short path |
| $t_{pcq}$ | Clock-to-Q propagation delay | property of the flip-flop |

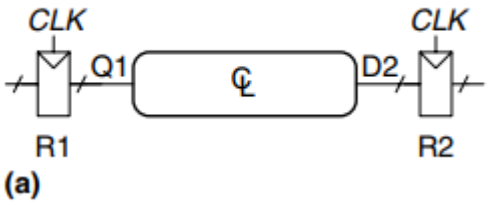
| Term | Name | Notes |
| --- | --- | --- |
| $t_{ccq}$ | Clock-to-Q contamination delay | property of the flip-flop |
| $t_{setup}$ | setup time | property of the flip-flop |
| $t_{hold}$ | hold time | property of the flip-flop |



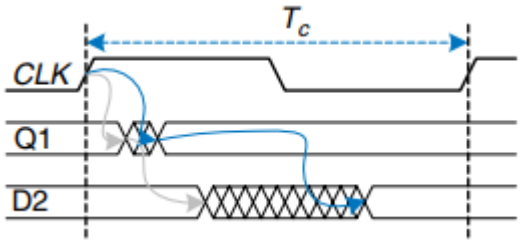

## Max-Delay Constraints

#### timing analyze

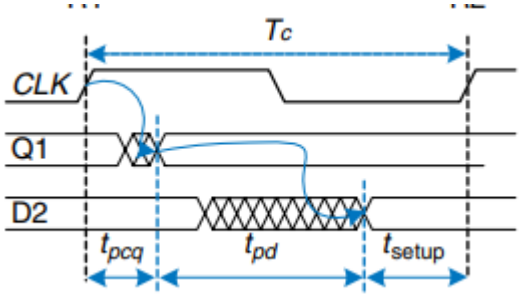
- on the rising edge of the clock, register R1 produces output $Q1$.
- These signals enter a block of combinational logic, producing $D2$, the input to register R2




- each output signal starts to change a contamination delay after its input change
- and settles to the final value within a propagation delay after its input settles
- gray arrows → contamination delay
- blue arrows → propagation delay




- showing **only** the **maximum** delay through the path
- to satisfy the setup time of R2, $D2$ must settle no later than the setup time before the next clock edge




- equation for the minimum clock period

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$
$$t_{pd} \leq T_c - (t_{pcd} + t_{setup})$$

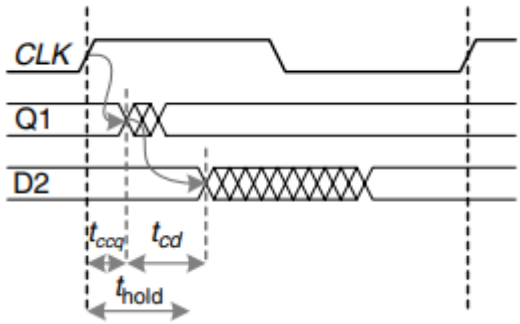
**design ramification**

- propagation delay of combinational logic is less than maximum
  - no problems
- otherwise?
  - flip-flop may sample wrong value or, worse, illegal value

- solutions: redesign combinational logic or increase clock period

## Min-Delay Constraints

- the register R2 has a **hold time constraint**
- its input $D2$, must **not** change until some time $t_{hold}$, after the rising edge of the clock

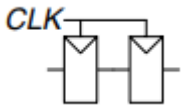



- equations

$$t_{ccq} + t_{cd} \geq t_{hold}$$
$$t_{cd} \geq t_{hold} - t_{ccq}$$

#### back-to-back flip-flops

- two flip-flops directly cascaded without causing hold time problems




- in such a case, $t_{cd} = 0$ ← there is no combinational logic between flip-flops

$$t_{hold} \leq t_{ccq}$$

- a reliable flip flop must have a hold time **shorter** than its contamination delay

#### design ramifications

- increasing the clock period does not fix hold time violations
  - redesign the combinational logic to increase its contamination delay
  - e.g. add buffers

# L10 Sequential Building Blocks

- terminology

| Term | Definition/ Description |
|---|---|
| Binary counter | a sequential arithmetic circuit with clock and reset inputs and an $N$-bit output |
| Shift register | has a clock, a serial input, a serial output and $N$ parallel outputs |
| Serial | |
| Parallel | |
| Serial-to-parallel | the input is provided serially at $S_{in}$, After $N$ cycles, the past $N$ inputs are available in parallel at $Q$ |
| Parallel-to-serial | loads $N$ bits in parallel, then shifts them out one at a time |

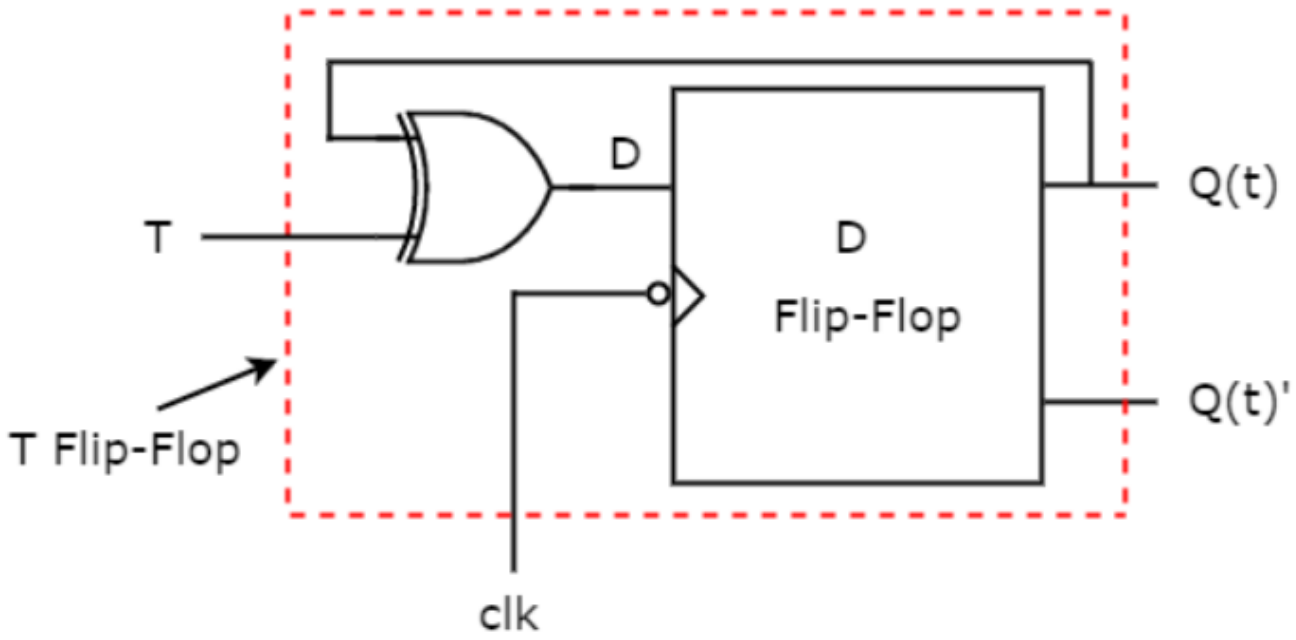
- shift register - modes of operation

| | |
|---|---|
| Serial-in, Parallel-out | Loaded with data one bit at a time; stored data is read all at once |
| Serial-in, Serial-out | Loaded with data one bit at a time; stored data is read one bit at a time |
| Parallel-in, Serial-out | Loaded with data in one cycle; stored data is read one bit at a time |
| Parallel-in, Parallel-out | Loaded with data in one cycle; stored data is read all at once |

## Binary Counter

**T flip-flop**

- T for toggle




| T flip-flop input | Present State | Next State | D flip-flop input |
|---|---|---|---|
| **T** | **Q$t$** | **Q$t+1$** | **D** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

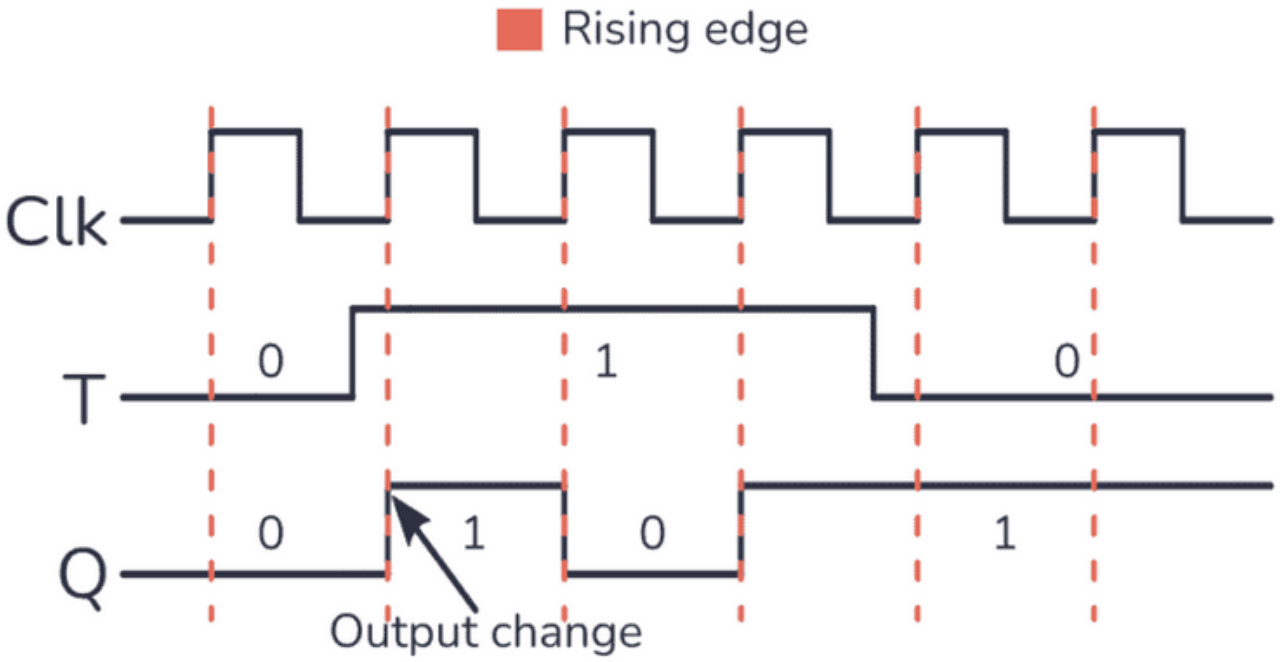
- patterns when counting up in binary
  - LSB toggles every cycle
  - middle bit - after the LSB is one in the previous cycle, the middle bit would change from the previous cycle (1→0 / 0→1)
    - e.g c1→c2, c3→c4, c5→c6

- MSB - after the middle bit and LSB both $1$, MSB would change
  - e.g. c3→c4

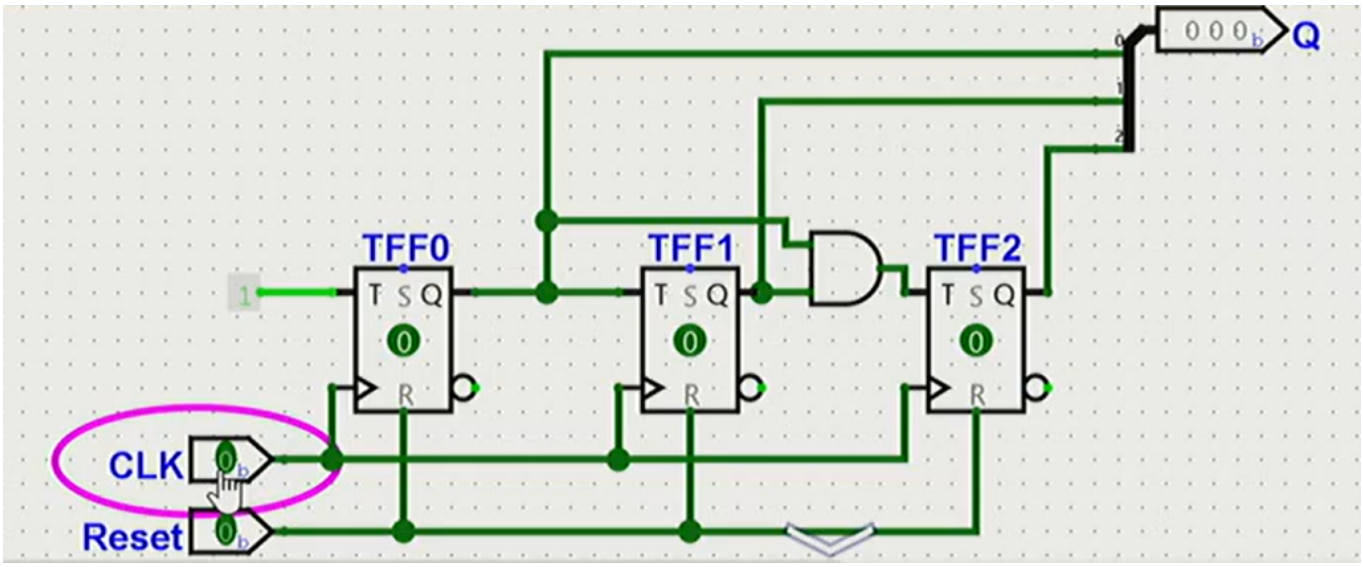
| Cycle | Binary |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

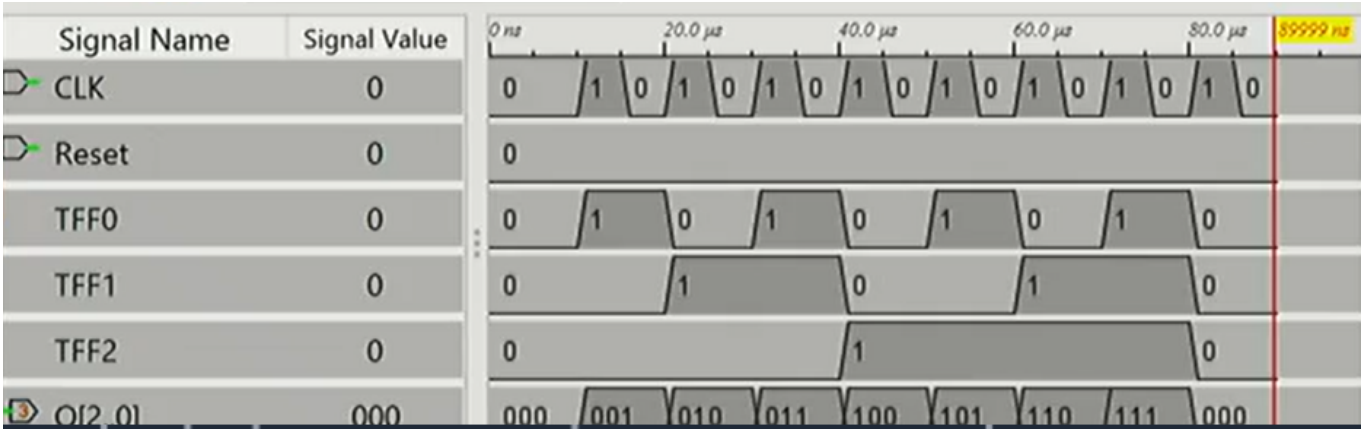
**Counting with T flip-flop**

- T connects to $V_{DD}$



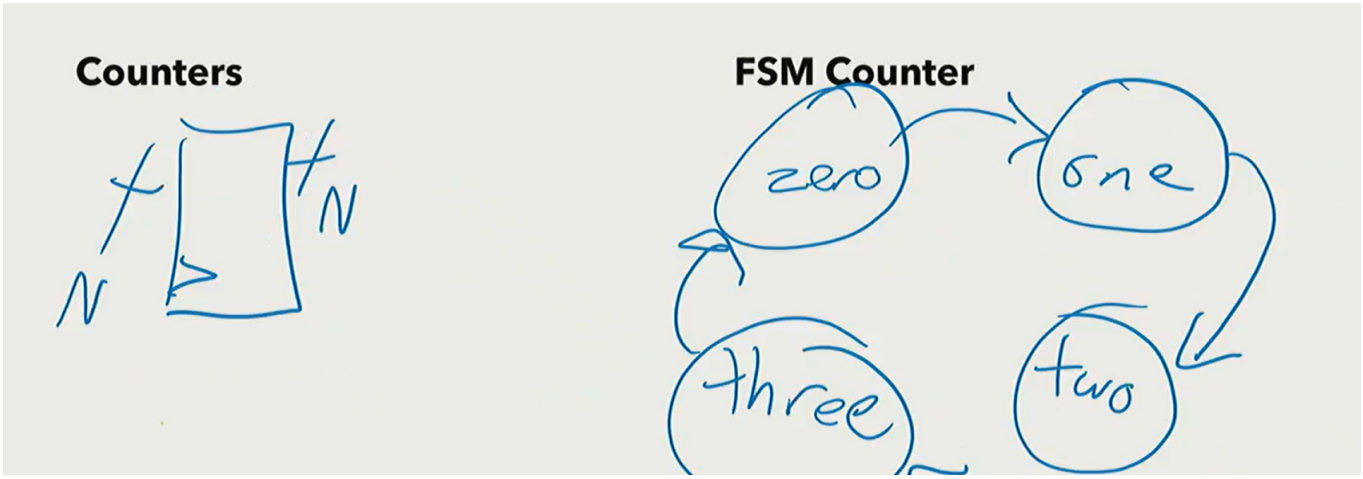

**A timing diagram for the T Flip-Flop**




**compare**

- counter with adder - area for the adder and propagation delay are high
- counter with T flip-flop - both are low



# L11 Synchronizers and Parallelism

## Metastability

#### terminology

| Term | Defintioin/ Description |
| --- | --- |
| Aperture time | the sum of the setup and hold time |
| Metastable | when a flip-flop samples an input that is changing during its aperture, the output $Q$ may momentarily take in the forbidden zone |
| Resolution time | a random time during the clock cycle, $t_{res}$ |
| Asynchronous input | |

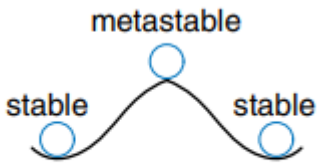
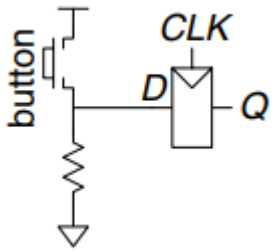



**Figure 3.49 Stable and metastable states**

- every bistable device has a metastable state bewteen the two stable states

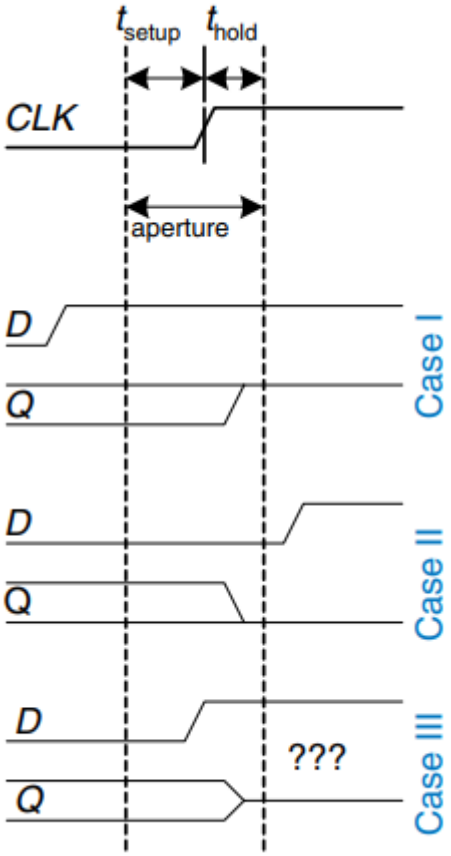
#### asynchronous inputs

- synchronous systems need aynchronous inputs
  - asynchronous inputs can laed to **metastable** valtages
  - should be to ensure that, given asynchronousinputs, the probability of enountering a metastable voltage is sufficiently small
- how to bring the input signal to "match" the timing of a system clock
  - when press the button, connect to the flip flop




**revolution time**

- case I - D goes HIGH **begore** aperture time
- case II - D goes HIGH **after** aperture time
- case III - D goes HIGH **during** aperture time

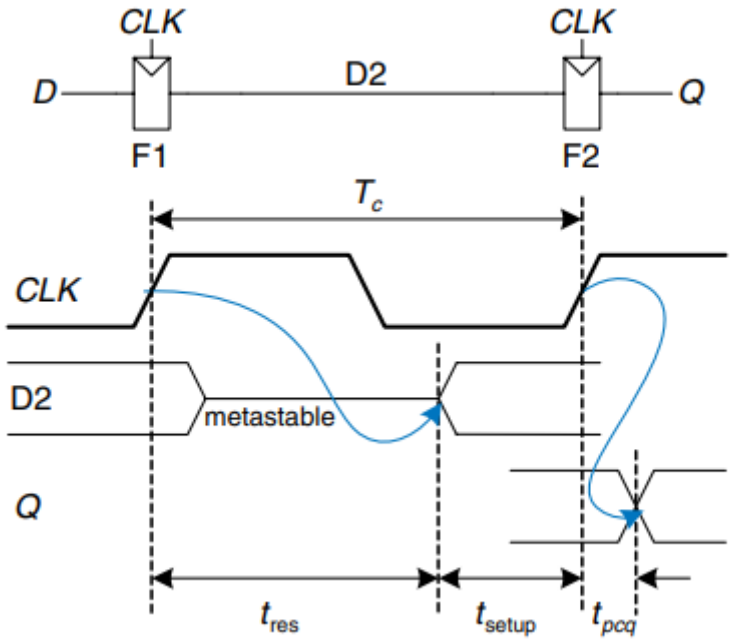



- probability that the resolution time <u>exceeds</u> some arbitrary time
  - $T_c$ - clock period
  - $T_0$ and $\tau$ are characteristic of the flip-flop
  - valid **only** for $t$ substantially **longer** than $t_{pcd}$

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}}$$

## Synchronizer

- receive asysnchronous input $D$ and a clock CLK; produce output $Q$
  - $Q$ has a valid logic level with extremely high probability
  - $D$ stable during the aperture → $Q$ same as $D$
  - $D$ change → $Q$ may take on either a HIGH or LOW value but **must not** be metastable
- if clock period is long enoughm $D2$ will resolve to a valid logic level before the end of the period with high probability




### failure

- synchronizer **fails** ← $Q$ becomes metastable

  → $t_{res} > T_c - t_{setup}$
- $D$ changes once per second

$$P(failure) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}}$$

- chanes N times per second

$$P(failure) = N \times \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}}$$

- system reliability is measured in *mean times between failures* (MTBF)

$$MTBF = \frac{1}{P(failure)/sec} = \frac{T_c e^{-\frac{T_c - t_{setup}}{\tau}}}{NT_0}$$

- When: $\tau = 200\ ps, T_0 = 150\ ps, t_{setup} = 500\ ps$
- Then: for MTBF to exceed 1 **year**
- $T_c$ equals approximately 3 nanoseconds (i.e., 333 MHz)

## Parallelism

- token - a group of inputs used to generate a group of outputs
- latency - **time** for one token to get from "start" to "finish"
- throughput - **the number of** tokens that can be produced per unit of time

**Example 3.14** COOKIE THROUGHPUT AND LATENCY

Ben Bitdiddle is throwing a milk and cookies party to celebrate the installation of his traffic light controller. It takes him 5 minutes to roll cookies and place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

**Solution:** In this example, a tray of cookies is a token. The latency is 1/3 hour per tray. The throughput is 3 trays/hour.

---

- spatial parallelism - **duplicate hardware** so that more than one task can be done at a time
- temporal parallelism
  - break a task up into **stages**
  - every task must pass through every stage
  - a different task will be in each stage at any one time → multi tasks can overlap

### Example 3.15 COOKIE PARALLELISM

Ben Bitdiddle has hundreds of friends coming to his party and needs to bake cookies faster. He is considering using spatial and/or temporal parallelism.

**Spatial Parallelism:** Ben asks Alyssa P. Hacker to help out. She has her own cookie tray and oven.

**Temporal Parallelism:** Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

**Solution:** The latency is the time required to complete one task from start to finish. In all cases, the latency is 1/3 hour. If Ben starts with no cookies, the latency is the time needed for him to produce the first cookie tray.

The throughput is the number of cookie trays per hour. With spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes. Hence, the throughput doubles, to 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, for a throughput of 4 trays/hour. These are illustrated in Figure 3.55.

If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

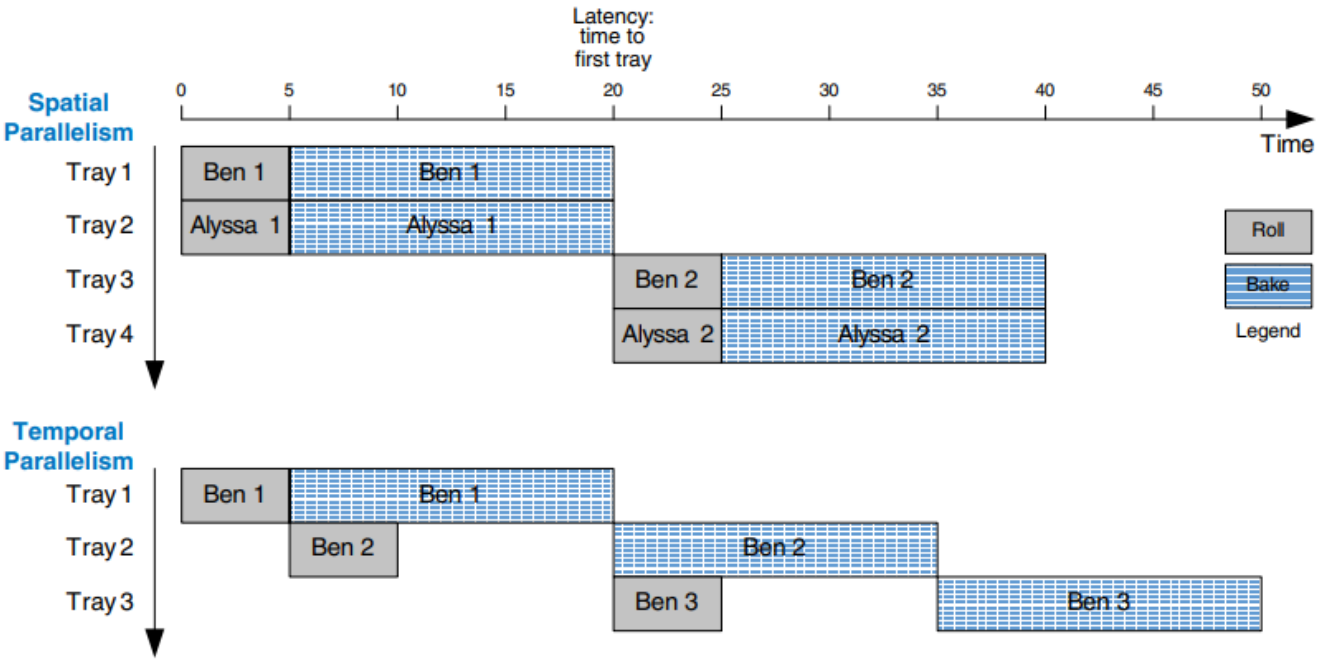



**Figure 3.55 Spatial and temporal parallelism in the cookie kitchen**

- pipelining - use **registers** between blocks of combinational logic instead of duplicate hardware

- assume $t_{pd} = 0,3ns$ and $t_{setup} = 0.2ns$

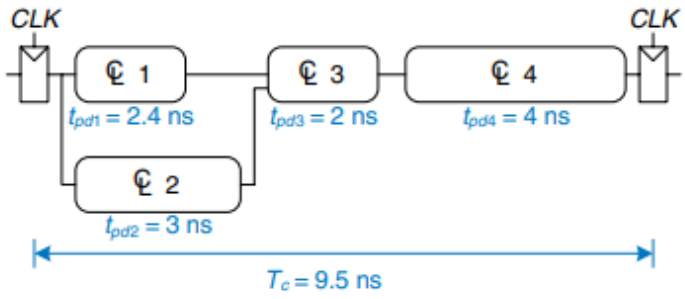



**Figure 3.56 Circuit with no pipelining**

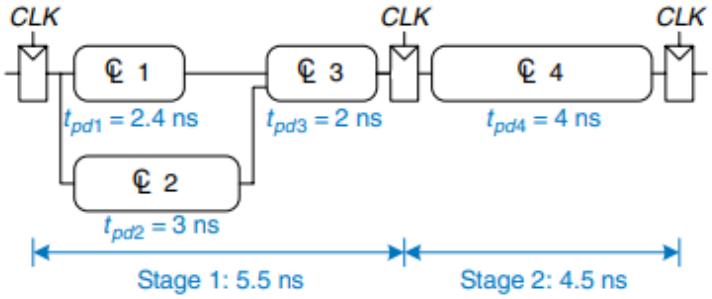



**Figure 3.57 Circuit with two-stage pipeline**

**Spatial parallelism**

- More area needed
- Multiple tasks completed independently

**Temporal parallelism**

- "Same" area
  - Note: some overhead due to pipelining/sequencing overhead
- Improves throughput very well, but some extra latency
  - i.e., from sequencing overhead
- What about dependencies?

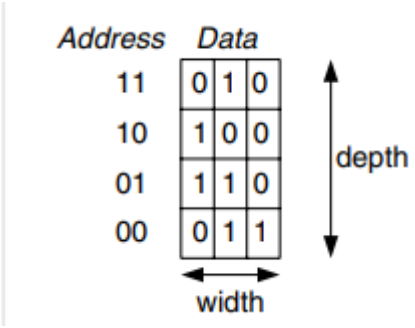
# L12 Memory

## Shared characteristics

### organization

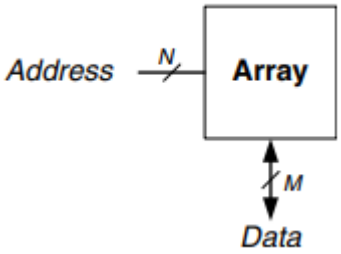
- memory is organized as a **two-dimensional** array of memory cells
- row - specified by an address
  - $N$-bit address → $2^N$ rows
- data - value read or written

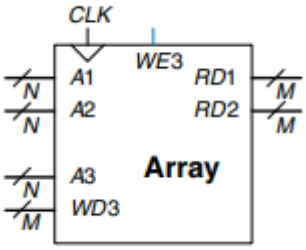
    - $M$-bit data → $M$ coloumns
- word - each row of data
    - $2^N M$ - bit words
- depth - #rows, width - #columns → depth $\times$ width




**interface**

- input - $N$-bit address
- input/output - $M$ - bit data
- operation - read/write



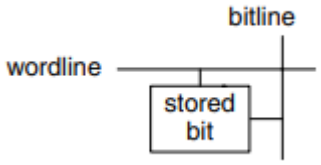




**bit cell**

built as an array of **bit cells**

- each stores $1$ bit of data
- wordline HIGH → **stored** bit transfers to bitline

- wordline LOW → bitline disconnected from the bit cell




- read
  - initial bitline → **left floating (Z)**
  - wordline turn ON → allow the store value to drive the bitline to $0$ or $1$
- write
  - bitline is strongly driven to the **desired value**
  - wordline turn ON → connecting the bitline to the stored bit
  - strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit

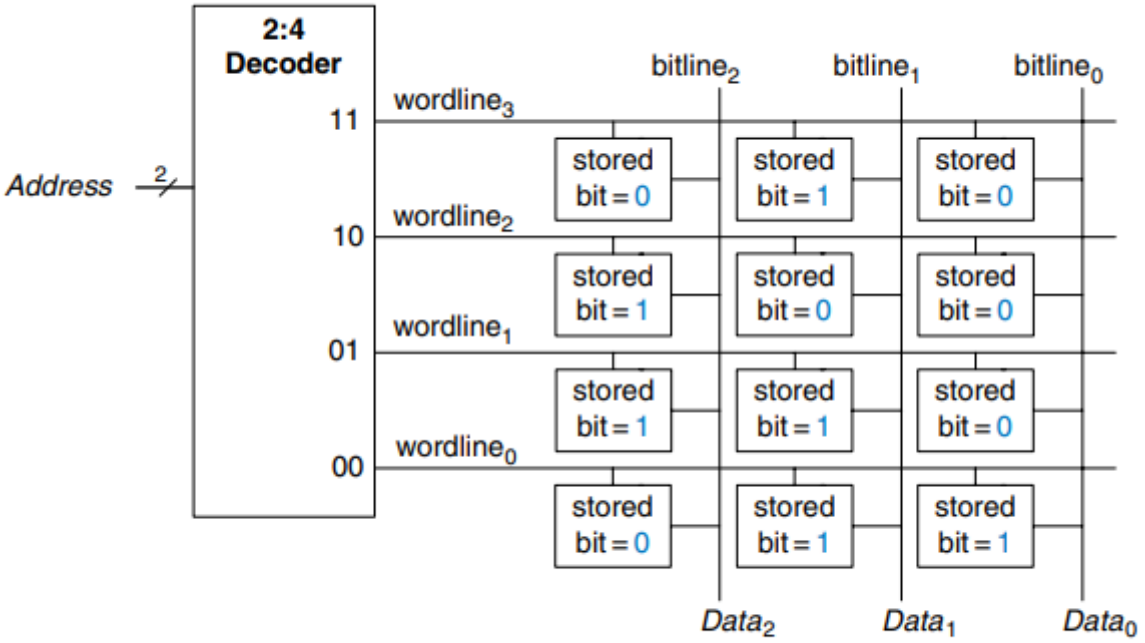



**Figure 5.42 4 × 3 memory array**

# Memory

- volatile ← RAM (random access memory)
  - **loses data** values when power is off
- Non-volatile ← ROM (read-only memory)
  - **retains data**, even without a power source

### DRAM

→ Dynamic Random Access Memory

- the bit value is stored in a **capacitor**
  - reading destroys the bit value
  - capacitors leak charge
- need to **restore** capacitor over time
  - on a read
  - on a refresh

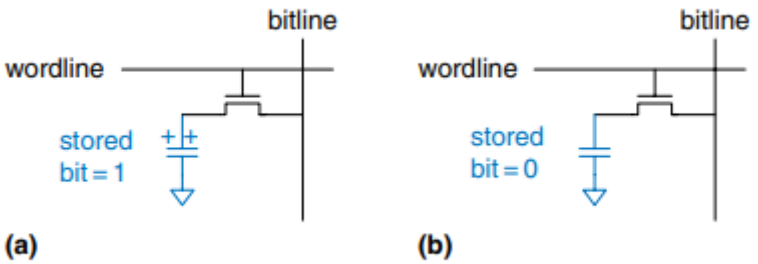



Figure 5.45 DRAM stored values

### SRAM

→ Static Random Access Memory

- the bit value is stored in **cross-coupled inverters** → do not need to be refreshed
- two outputs → bitline & inverse of bitline

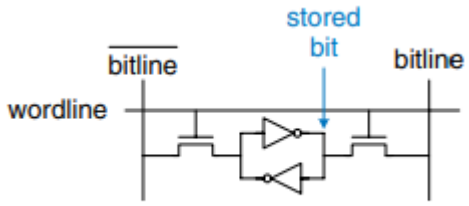



**Figure 5.46 SRAM bit cell**

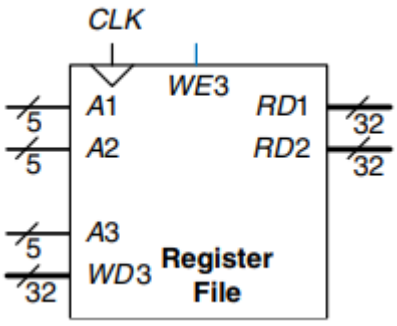
### Comparison

| Memory Type | Transistors per Bit Cell | Latency |
|---|---|---|
| flip-flop | ~20 | fast |
| SRAM | 6 | medium |
| DRAM | 1 | slow |

- are all volatile memories
- flip-fops → found in the **datapath** and **control unit**
- SRAM → found in **caches**, useful for **register files**
- DRAM → main memory (high density), Off chip

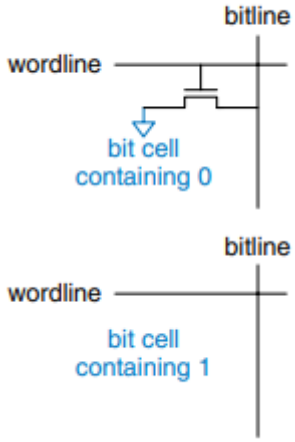
### Register Files

- built as a small, multi-ported memory array
  - can use registers
  - typically use SRAM
- datapath is made up of many registers
- store a large group of registers in a register file

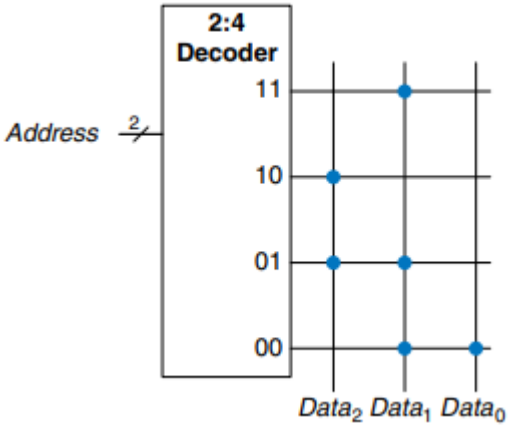


**ROM**

- stores a bit as the **presence** or **absence** of a transistor
  - weakly pull bitline high
  - assert wordline




- dot notation - indicate a data bit is **1** using a dot




# L13 Assembly Instruction

**terminology**

| Term | Definition |
| --- | --- |
| Mnemonic | indicates what operation to perform |
| Source operand | the variable where the operation is performed |

| Term | Definition |
| --- | --- |
| Destination operand | the varaible where the result is written |
| RISC | reduced instruction set computer |
| CISC | complex instruction set computers |

**High-Level Code**

```
a = b + c - d;    // single-line comment
                  /* multiple-line
                     comment */
```

**MIPS Assembly Code**

```
sub t, c, d       # t = c - d
add a, b, t       # a = b + t
```

**design principle**

- simpilicity favors regularity → `add` and `sub` have a consistent format
- make the common case fast → hardware for `add` and `sub` is simple
- smaller is faster
  - fewer the registers, the faster they can be accessed
- good design demands good compromise

## Registers

- instructions need to access operands quickly so that they can run fast → specify a smalle number of **registers** that hold commonly used operands
- register set/file - **32** registers that MIPS architecture uses
  - each register stores **32** bits
- **saved** register - register names beginning with `$s`
  - special connation → used with procedure calls
- **temporary** register - register names beginning with `$t`
  - used for storing temporary variables

| High-Level Code | MIPS Assembly Code |
| --- | --- |
| `a = b + c - d;` | `# $s0 = a, $s1 = b, $s2 = c, $s3 = d`<br><br>`sub $t0, $s2, $s3  # t = c - d`<br>`add $s0, $s1, $t0  # a = b + t` |

**Table 6.1 MIPS register set**

| Name | Number | Use |
| --- | --- | --- |
| $0 | 0 | the constant value 0 |
| $at | 1 | assembler temporary |
| $v0-$v1 | 2–3 | procedure return values |
| $a0-$a3 | 4–7 | procedure arguments |
| $t0-$t7 | 8–15 | temporary variables |
| $s0-$s7 | 16–23 | saved variables |
| $t8-$t9 | 24–25 | temporary variables |
| $k0-$k1 | 26–27 | operating system (OS) temporaries |
| $gp | 28 | global pointer |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | procedure return address |

## Constants

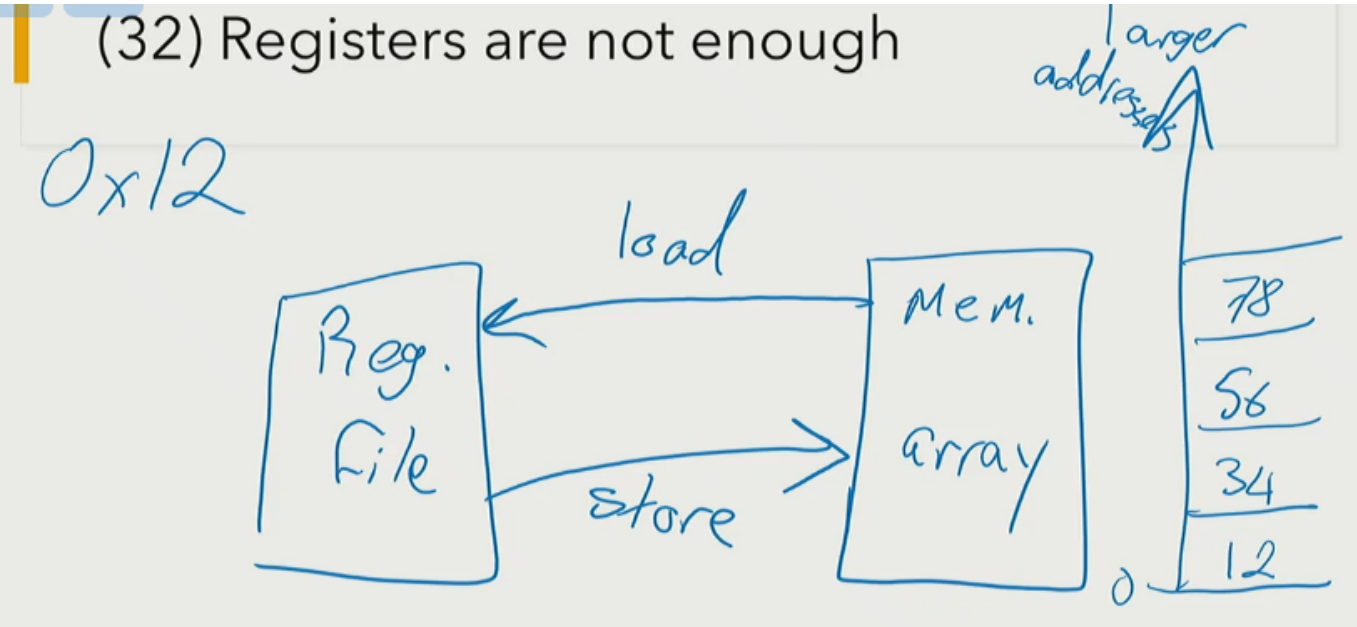
- **immediate** - constants
  - their values are immediately available from the instruction
  - **do not** require a register or memory access
- `addi` - adds the immediate specified in the instruction to a value in a registers

**Code Example 6.9** IMMEDIATE OPERANDS

| High-Level Code | MIPS Assembly Code |
| --- | --- |
| `a = a + 4;`<br>`b = a - 12;` | `# $s0 = a, $s1 = b`<br>`addi $s0, $s0, 4 # a = a + 4`<br>`addi $s1, $s0, -12 # b = a - 12` |

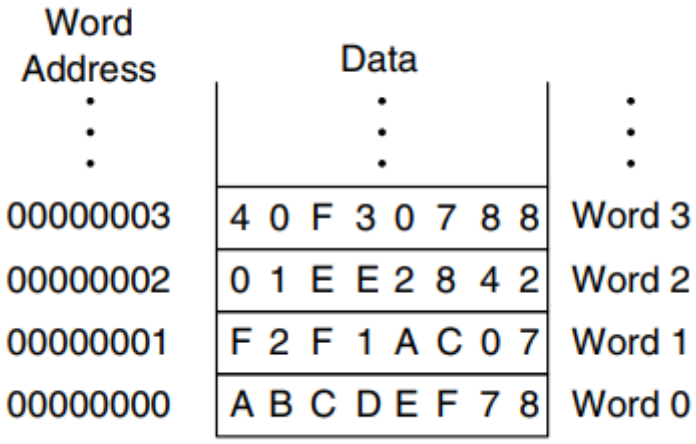
- the immediate value is a part of the instruction itself
  - space in the instruction are reserved for an immediate value
- MIPS - immediates are 16-bit two's complement numbers
  - [-32768, 32767]

## Memory




**word-addressable memory**

- each 32-bit data word has a unique 32-bit address
- both the 32-bit word address and the 32-bit data value are written in **hexadecimal**
- memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top




**load word instruction**

- `lw` - read a data word from memory into a register
- specifies the **effective address** in memory as the **sum** of a **base address** and an **offset**.
  - base address (`$0`) - register, offset (`1`)- constant
  - destination operand (`$s3`)
  - `lw` reads from memory address `($0+1)=1` → `$s3` holds the value 0xF2F1AC07

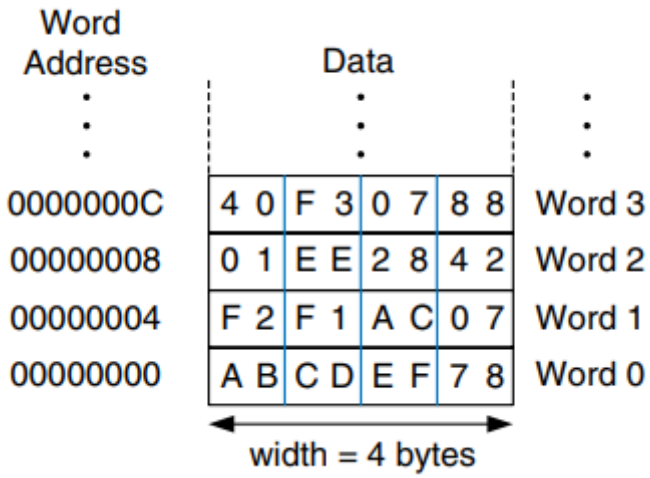
**Assembly Code**

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

- store word instruction `sw`
  - write a date word from a register into memory

**byte-addressable memory**

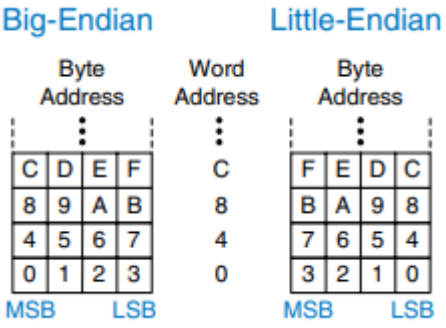
- MIPS memory model is byte-addressable, **not** word-addressable
  - each data byte has a unique address
  - one 32-bit word = four 8-bit bytes




**MIPS Assembly Code**

```
lw $s0, 0($0)          # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)          # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)        # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)          # write $s3 to data word 1
sw $s4, 0x20($0)       # write $s4 to data word 8
sw $s5, 400($0)        # write $s5 to data word 100
```

- when using `lw` and `sw` - addresses must be **word aligned**
  - the word size is 32 bits = 4 bytes
  - and address that is word aligned is divisible by 4
- **big-endian & little-endian**
  - both - MSB: left, LSB: right
  - different - big/little numbered starting with 0 at **MSB/LSB** end



**Example 6.2** BIG- AND LITTLE-ENDIAN MEMORY

Suppose that `$s0` initially contains 0x23456789. After the following program is run on a big-endian system, what value does `$s0` contain? In a little-endian system? `lb $s0, 1($0)` loads the data at byte address `(1 + $0) = 1` into the least significant byte of `$s0`. `lb` is discussed in detail in Section 6.4.5.

```
sw $s0, 0($0)
lb $s0, 1($0)
```

**Solution:** Figure 6.4 shows how big- and little-endian machines store the value 0x23456789 in memory word 0. After the load byte instruction, `lb $s0, 1($0)`, `$s0` would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.


**Figure 6.4** Big-endian and little-endian data storage

# L14 Machine Code

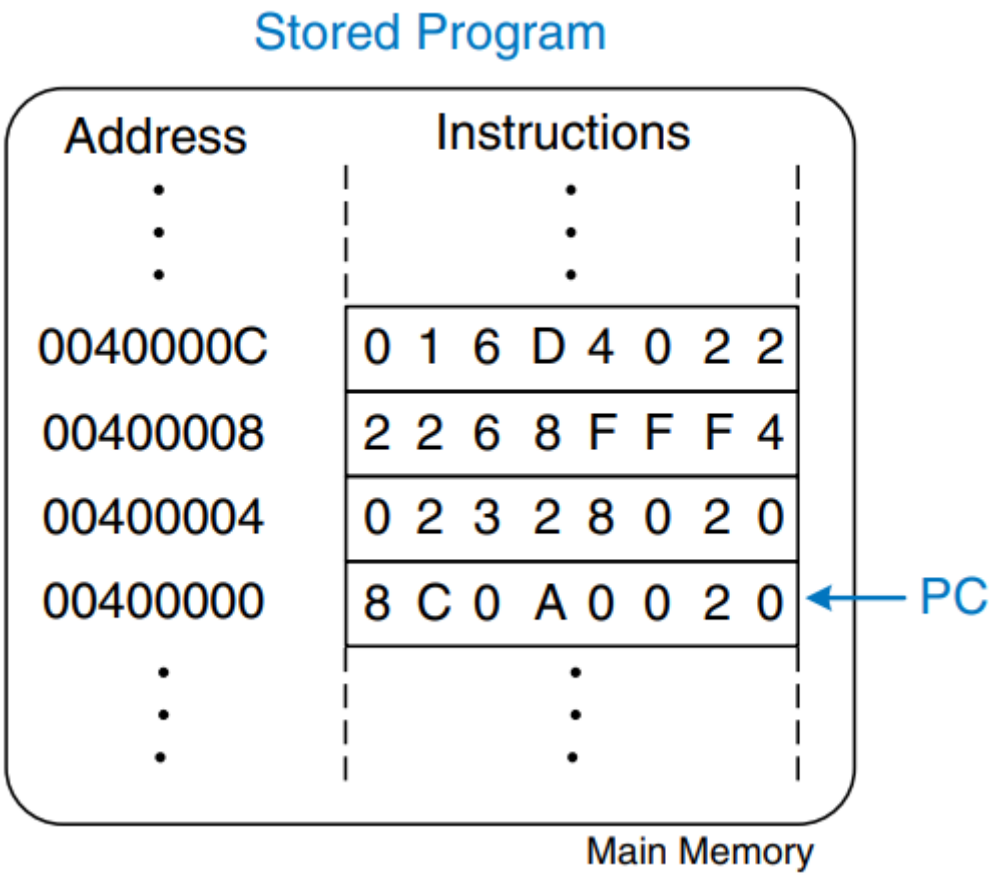
## program

- is made up of (lots and lots of) bits
- instructions are N-bits numbers that encode
  - the operation to perform (X bit)
  - other details needed (N-X bits) (e.g. operands)
- assembly language - human-readable format of instructions
- machine language - computer-readable format of instructions

#### stored in memory

- advantage
  - can run different programs easily
    - no hardware reconfiguration or rewriting

    - general purpose computing
- disadvantage - instructions need to be fetched from memory into the processor
- PC - program counter
    - instructions start at address **0×00400000**
    - on each cycle, PC=PC+4




## Instruction Formats

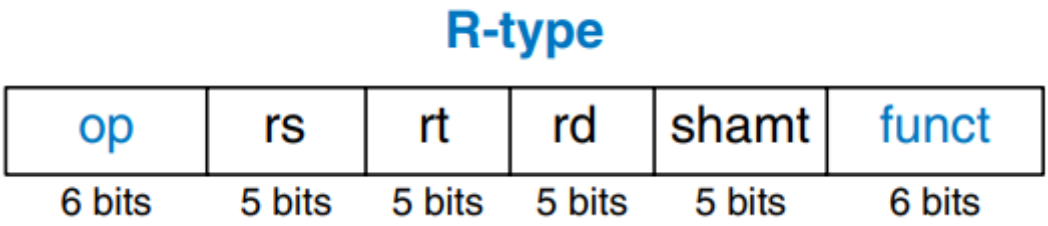
#### shared characteristics

- all instructions are 32 bits
- the **most-significant 6 bits** are the operation code

#### R-type

- register-type
- use three registers as operands - two as sources, one as a destination
    - `rs`, `rt`, `rd`

- all R-type instructions have an **opcode of 0**
- `funct` - the R-type operation
- `shamt` - only for **shift** operations




- instructions
  - [`rd`] - write to the register address
  - [`rs`] - read from the register address

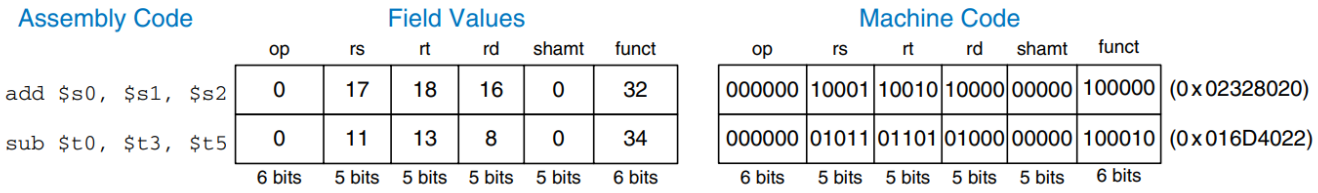
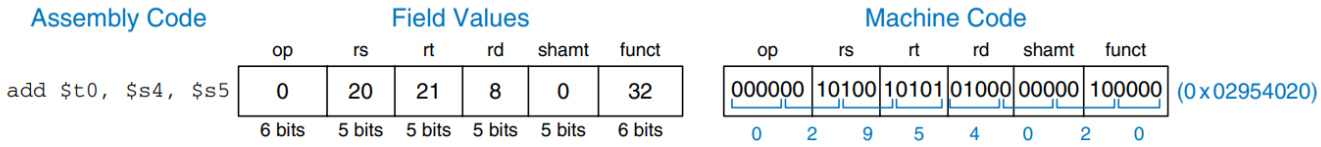


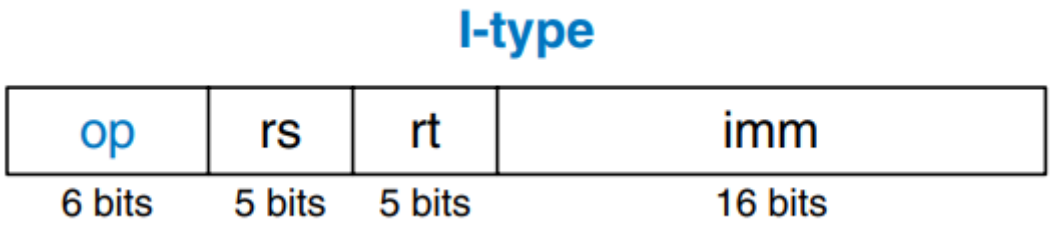

**Figure 6.6 Machine code for R-type instructions**




| funct | mnemonic | description | operation |
|---|---|---|---|
| 100000 (32) | add | Add | [rd] = [rs] + [rt] |
| 100010 (34) | sub | Subtract | [rd] = [rs] - [rt] |

### I-type

- immediate-type
- two register operands and on immediate operand
  - `rs`, `rt`, `imm`
- `imm` - holds the 16-bit immediate
- `SignImm` - sign extend `imm`



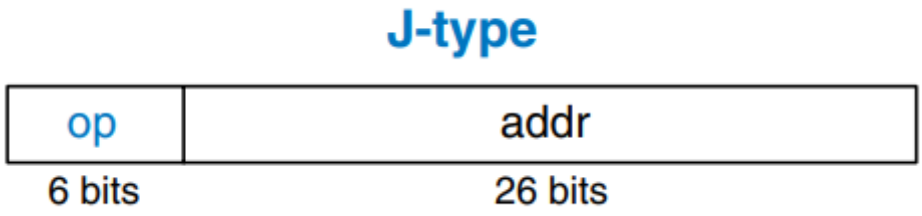

| Assembly Code | op | rs | rt | imm | Machine Code | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | op | rs | rt | imm | op | rs | rt | imm | |
| addi $s0, $s1, 5 | 8 | 17 | 16 | 5 | 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x22300005) |
| addi $t0, $s3, -12 | 8 | 19 | 8 | –12 | 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF4) |
| lw $t2, 32($0) | 35 | 0 | 10 | 32 | 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A0020) |
| sw $s1, 4($t1) | 43 | 9 | 17 | 4 | 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD310004) |
| | 6 bits | 5 bits | 5 bits | 16 bits | 6 bits | 5 bits | 5 bits | 16 bits | |

**Figure 6.9** **Machine code for I-type instructions**

| opcode | mnemonic | description | operation |
|---|---|---|---|
| 001000 (8) | addi | Add immediate | [rt] = [rs] + SignImm |
| 100011 (35) | lw | Load word | [rt] = [Address] |
| 101011 (43) | sw | Store word | [Address] = [rt] |

### J-type

- jump-type
- **no** operands
- 26 bits for an **address**



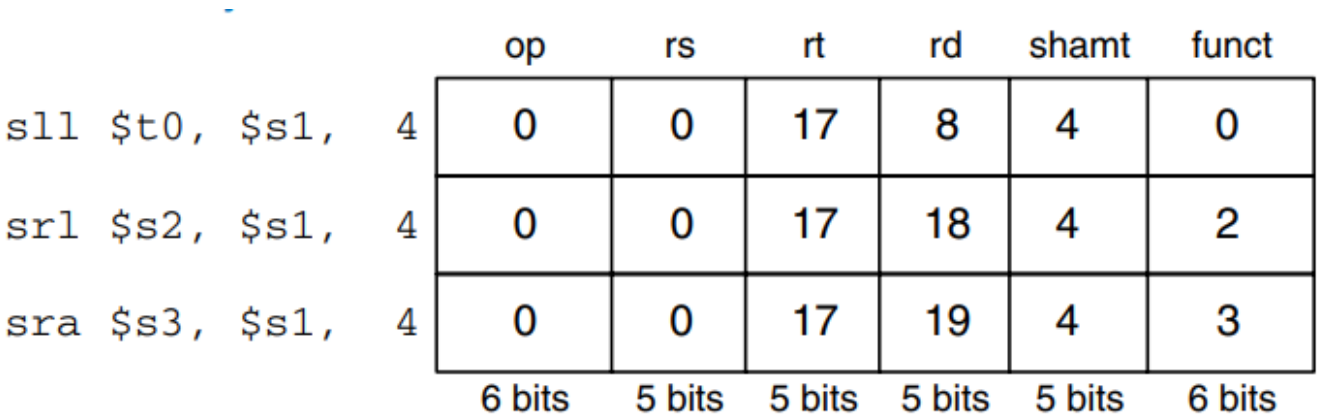

# L15 Branch Instruction

# Arithmetic/logical instruction

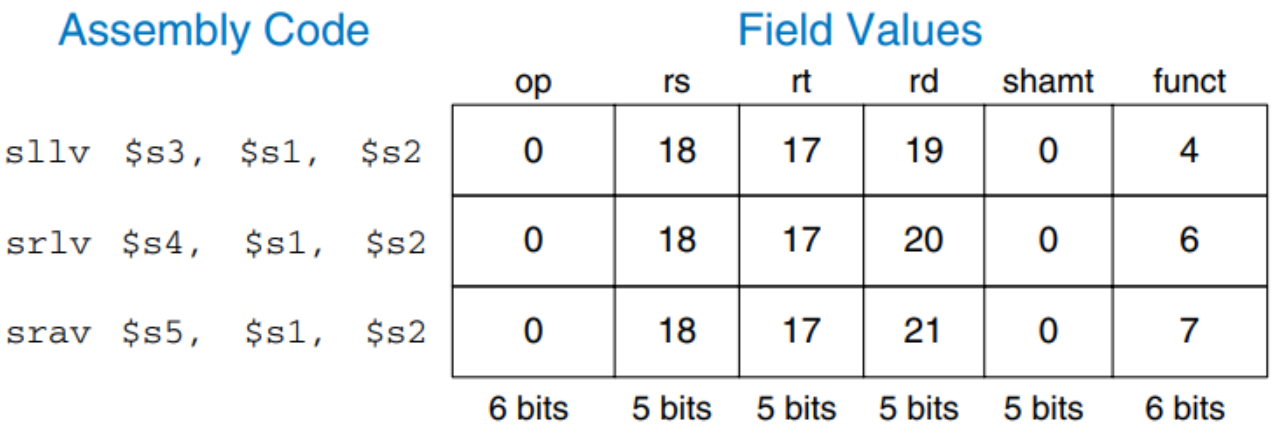
### terminology

| Mnemonic | Example usage |
|---|---|
| and / andi | `0xFFFF0000 AND 0x46A1F0B7 = 0x46A10000` masking |
| or/ ori | `0x347A0000 OR 0x000072FC = 0x347A72FC` combining |
| xor/ xori | |
| nor | |
| sll/ sllv | shift left logical (variable) |
| srl/ srlv | shift right logical (variable) |
| sra/ srav | shift right arithmetic (variable) |

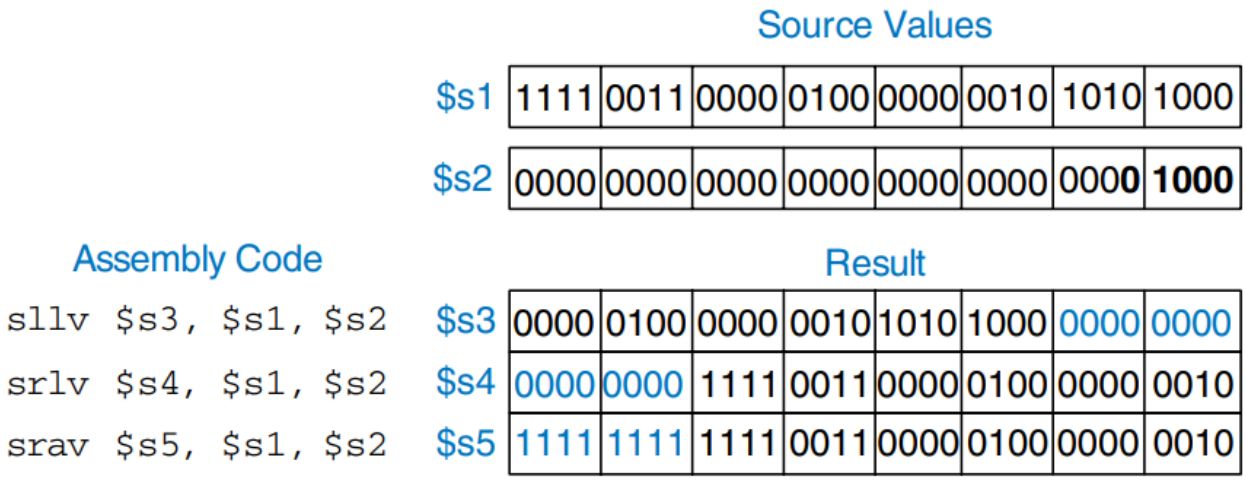
- `rt` holds the value to be shift



| | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| `sll $t0, $s1, 4` | 0 | 0 | 17 | 8 | 4 | 0 |
| `srl $s2, $s1, 4` | 0 | 0 | 17 | 18 | 4 | 2 |
| `sra $s3, $s1, 4` | 0 | 0 | 17 | 19 | 4 | 3 |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |



| Assembly Code | Field Values | | | | | |
|---|---|---|---|---|---|---|
| | op | rs | rt | rd | shamt | funct |
| `sllv $s3, $s1, $s2` | 0 | 18 | 17 | 19 | 0 | 4 |
| `srlv $s4, $s1, $s2` | 0 | 18 | 17 | 20 | 0 | 6 |
| `srav $s5, $s1, $s2` | 0 | 18 | 17 | 21 | 0 | 7 |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |




### generating constant

- `lui` - load upper immediate
- `ori` - load into the lower half

```
# 16-bit constant
addi $s0, $0, 0x43fc


# 32-bit constant

# $s0 = 0x65de0000
lui $s0, 0x6d5e

# s0 = 0x65de43fc
ori $s0, $0, 0x43fc
```

## Branching

### conditional branching

- `beq` - branch if **equal** → `beq $rs, $rt, imm`
  - PC =BTA → branch is taken
- `bne` - branch if **not equal** → `bne $rs, $rt, imm`

    - PC = PC + 4 → branch is **not** taken
- branch target address (**BTA**)
    - branch delay slot
        - BTA = PC + 4 +(SignImm << 2) → SignImm shift left 2 → **mutiple by 4**
    - no branch delay slot
        - BTA = PC + (SignImm << 2)
- **label** - indicates a memory location
    - are translated into instruction addresses
    - `<lable_name>:`

| Address | Label | Instruction |
|---|---|---|
| 0x400000 | | `add $s0, $0, $0` |
| 0x400004 | | `beq $s0, $s0, target` |
| 0x400008 | | ... |
| 0x40000C | | ... |
| 0x400010 | target: | |
| 0x400014 | | <some instruction> |
| 0x400018 | | ... |
| 0x40001c | | ... |

```
    addi $s0, $0, 4 # $s0 = 0 + 4 = 4
    addi $s1, $0, 1 # $s1 = 0 + 1 = 1
    s11 $s1, $s1, 2 # $s1 = 1 << 2 = 4
    bne $s0, $s1, target # $s0 == $s1, so branch is not taken
    addi $s1, $s1, 1 # $s1 = 4 + 1 = 5
    sub $s1, $s1, $s0 # $s1 = 5 - 4 = 1

target:
    add $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```

**unconditional branching**

- jump - `j target`
  - jumps directly to the instruction at the specified label.

```
    addi $s0, $0, 4 # $s0 = 4
    addi $s1, $0, 1 # $s1 = 1
    j target # jump to target
    addi $s1, $s1, 1 # not executed
    sub $s1, $s1, $s0 # not executed

target:
    add $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```

- JTA calculation
  - most significant $4$ bits from **(PC + 4)**
  - **26 bits from instruction**
  - least significant 2 bits are 0

## BTA "distance"

- Immediate is an "offset"
- Immediate can be positive or negative
  - Positive: branching "forward"
  - Negative: branching "backward"
- What is the maximum distance?

## JTA "distance"

- $JTA = \begin{cases} (PC + 4)[31:28] \\ addr \\ 2'b0 \end{cases}$
- or, equivalently,
- $JTA = (PC \ \& \ 0xF0000000) \mid (addr \ll 2)$
- The immediate is *not* an offset
  - It is a part of the final computed address

# Conditional Statements

### if statement

- high level

```
if (i == j):
    f = g + h;

f = f + i;
```

- assembly

```
    # $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
    bne $s3, $s4, L1 # if i ! = j, skip if block
    add $s0, $s1, $s2 # if block: f = g + h

L1:
    sub $s0, $s0, $s3 # f = f + i
```

#### if-else statement

- high level

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

- assembly

```
    # $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
    bne $s3, $s4, else # if i ! = j, branch to else
    add $s0, $s1, $s2 # if block: f = g + h
    j L2 # skip past the else block

else:
    sub $s0, $s0, $s3 # else block: f = f - i

L2:
```

# L16 Loops and Arrays

## Loop

#### while loop

- high level

```
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

- assembly

```
    # $s0 = pow, $s1 = x
    addi $s0, $0, 1 # pow = 1
    addi $s1, $0, 0 # x = 0

    addi $t0, $0, 128 # t0 = 128 for comparison

while:
    beq $s0, $t0, done # if pow == 128, exit while
    sll $s0, $s0, 1 # pow = pow * 2
    addi $s1, $s1, 1 # x = x + 1
    j while
done:
```

**for loop**

- high level

```
int sum = 0;

for (i = 0; i != 10; i = i + 1) {
    sum = sum + i;
}
```

- assembly

```
    # $s0 = i, $s1 = sum
    add $s1, $0, $0 # sum = 0
    addi $s0, $0, 0 # i = 0
    addi $t0, $0, 10 # $t0 = 10

for:
    beq $s0, $t0, done # if i == 10, branch to done
    add $s1, $s1, $s0 # sum = sum + i
    addi $s0, $s0, 1 # increment i
    j for

done:
```

## Magnitude Comparison

| funct | mnemonic | description | operation |
|---|---|---|---|
| 101010 (42) | slt | set less than | [rs] < [rt] ? [rd] = 1 : [rd] = 0 |

```
if [rs] < [rt]:
    [rd] = 1
else:
    [rd] = 0
```

- high level

```
// high-level code
int sum = 0;

for (i = 1; i < 101; i = i * 2)
    sum = sum + i;
```

- assembly

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0 # sum = 0
    addi $s0, $0, 1 # i = 1
    addi $t0, $0, 101 # $t0 = 101
loop:
    slt $t1, $s0, $t0 # if (i < 101) $t1 = 1, else $t1 = 0
    beq $t1, $0, done # if $t1 == 0 (i >= 101), branch to done
    add $s1, $s1, $s0 # sum = sum + i
    sll $s0, $s0, 1 # i = i * 2
    j loop
done:
```

## Arrays

### Assembler directives

- Programs are divided into segments
  - text segment: code
  - data segment: data
- Assembler directives are like instructions for the assembler (not the hardware)
- Directive: .text
  - The lines after this directive are instructions
  - Generally, start at 0x00400000
- Directive: .data
  - The lines after this directive are data
  - Generally, start at 0x10000000
- Directive: .word
  - A sequence of data, where each element in the sequence requires word-size bytes

```
.data 0x100070000
    .word 0x4, 0xc, 0x2, 0x5, 0x1

.text
    # load the base address
```

```
lui $s0, 0x1000
ori $s0, $s0, 0x7000

# x = array[0]
lw $t0, 0($s0)

# y = array[1]
lw $t1, 4($s0)

addi $t2, $0, 0
```

**High-Level Code**

```
int array [5];


array[0] = array[0] * 8;


array[1] = array[1] * 8;
```

**MIPS Assembly Code**

```
# $s0 = base address of array
  lui    $s0, 0x1000       # $s0 = 0x10000000
  ori    $s0, $s0, 0x7000  # $s0 = 0x10007000

  lw     $t1, 0($s0)       # $t1 = array[0]
  sll    $t1, $t1, 3       # $t1 = $t1 << 3 = $t1 * 8
  sw     $t1, 0($s0)       # array[0] = $t1

  lw     $t1, 4($s0)       # $t1 = array[1]
  sll    $t1, $t1, 3       # $t1 = $t1 << 3 = $t1 * 8
  sw     $t1, 4($s0)       # array[1] = $t1
```

**High-Level Code**

```
int i;
int array[1000];


for (i=0; i < 1000; i = i + 1) {


  array[i] = array[i] * 8;


}
```

**MIPS Assembly Code**

```
# $s0 = array base address, $s1 = i
# initialization code
  lui    $s0, 0x23B8       # $s0 = 0x23B80000
  ori    $s0, $s0, 0xF000  # $s0 = 0x23B8F000
  addi   $s1, $0           # i = 0
  addi   $t2, $0, 1000     # $t2 = 1000

loop:
  slt    $t0, $s1, $t2     # i < 1000?
  beq    $t0, $0, done     # if not then done
  sll    $t0, $s1, 2       # $t0 = i * 4 (byte offset)
  add    $t0, $t0, $s0     # address of array[i]
  lw     $t1, 0($t0)       # $t1 = array[i]
  sll    $t1, $t1, 3       # $t1 = array[i] * 8
  sw     $t1, 0($t0)       # array[i] = array[i] * 8
  addi   $s1, $s1, 1       # i = i + 1
  j      loop              # repeat
done:
```

# L17 Procedure

## Procedure Calls

**terminology**

| term | description |
| --- | --- |
| caller | the calling procedure |
| callee | the called procudure |
| arguments | input |
| return value | output |
| return address | where to return to after it completes |

| opcode / funct | mnemonic | description | operation |
| --- | --- | --- | --- |
| 000011 / (N/A) | jal | Jump and link | $ra = PC + 4<br>PC = JTA |
| 000000 / 001000 | jr | Jump register | PC = [rs] |

**High-Level Code**

```
int main() {
  simple();
  . . .
}
// void means the function returns no value
void simple() {
  return;
}
```

**MIPS Assembly Code**

```
0x00400200 main:   jal simple # call procedure
0x00400204         . . .


0x00401020 simple: jr $ra     # return
```

**input & output**

procedures use $a0$–a3 for input arguments and $v0$–v1 for the return value

- input - use `$a0` to `$a3` → 4 register, 128 bits
- output - use `$v0` and `$v1` → 2 register, 64 bits

**High-Level Code**

```
int main ()
{
  int y;

   . . .

  y = diffofsums (2, 3, 4, 5);

   . . .
}

int diffofsums (int f, int g, int h, int i)
{
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

**MIPS Assembly Code**

```
# $s0 = y

main:
  . . .
  addi $a0, $0, 2   # argument 0 = 2
  addi $a1, $0, 3   # argument 1 = 3
  addi $a2, $0, 4   # argument 2 = 4
  addi $a3, $0, 5   # argument 3 = 5
  jal diffofsums    # call procedure
  add $s0, $v0, $0  # y = returned value
  . . .

# $s0 = result
diffofsums:
  add $t0, $a0, $a1 # $t0 = f + g
  add $t1, $a2, $a3 # $t1 = h + i
  sub $s0, $t0, $t1 # result = (f + g) - (h + i)
  add $v0, $s0, $0  # put return value in $v0
  jr  $ra           # return to caller
```

**High-level code**

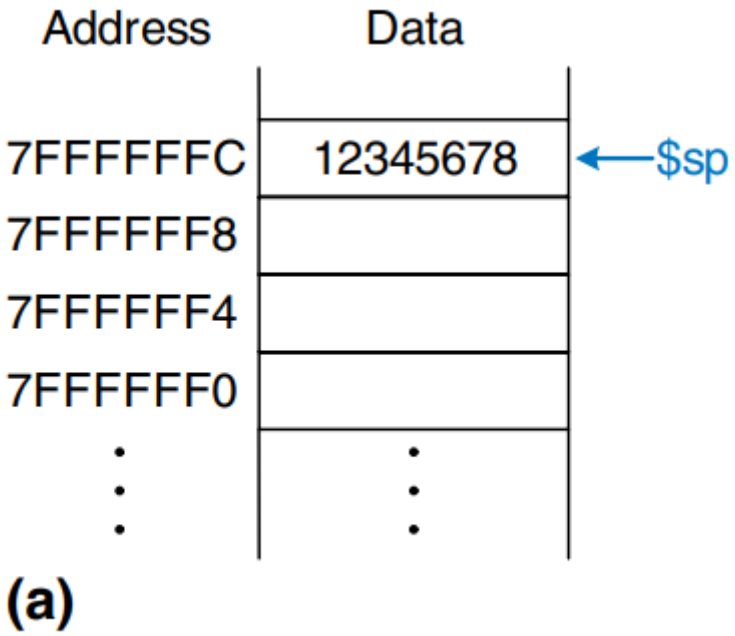
```
def diffofsums(f: int, g: int, h: int, i: int) -> int:
    result = (f + g) - (h + i)
    return result

diffofsums(2, 3, 4, 5)
```

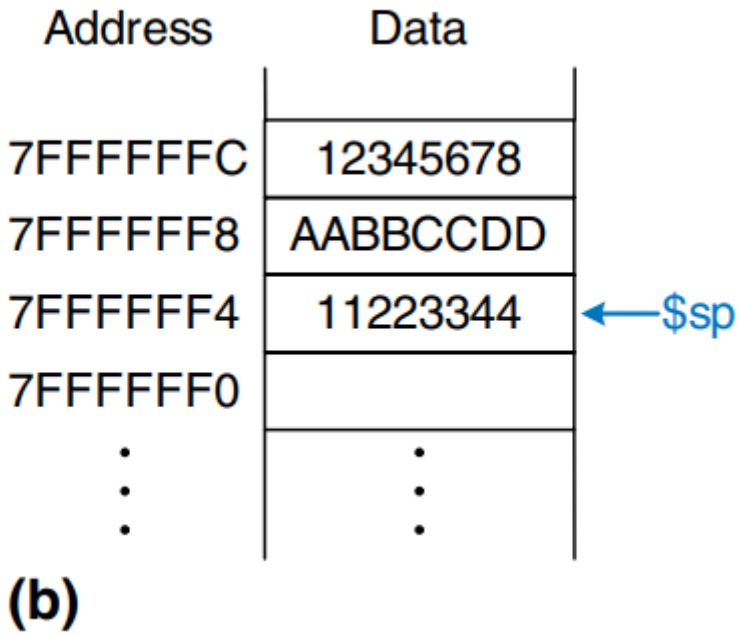
**MIPS assembly code**

```
.text
    ...
    jal diffofsums
    ...
diffofsums:
    add $t0, $a0, $a1
    add $t1, $a2, $a3
    sub $a0, $t0, $t1
    add $v0, $a0, $0
    jr $ra
```

## Stack

- a last-in-first-out queue in memory
- the stack grows **down** → expands to a **smaller** memory address
- used to store **local variables**
- `$sp` - points to the **top** of the stack
  - **cannot** access memory below `$sp`




- should not modify any registers besides the one containing the return value, `$t0`
- steps

1. Makes space on the stack to store the values of one or more registers.
2. Stores the values of the registers on the stack.
3. Executes the procedure using the registers.
4. Restores the original values of the registers from the stack.
5. Deallocates space on the stack.

**Before (no saving)**

```
diffofsums:
        # BODY
        add $t0, $a0, $a1
        add $t1, $a2, $a3
        sub $s0, $t0, $t1
        add $v0, $s0, $0

        jr $ra
```

**After**

```
diffofsums:
        # PROLOGUE
        addi $sp, $sp, -12
        sw $s0, 8($sp)
        sw $t0, 4($sp)
        sw $t1, 0($sp)

        # BODY
```

- restoring register values

**Before (no restoring)**

```
diffofsums:
        # BODY
        add $t0, $a0, $a1
        add $t1, $a2, $a3
        sub $s0, $t0, $t1
        add $v0, $s0, $0

        jr $ra
```

**After**

```
diffofsums:
        # PROLOGUE then BODY
        # EPILOGUE
        lw $t1, 0($sp)
        lw $t0, 4($sp)
        lw $s0, 8($sp)
        addi $sp, $sp, 12
        jr $ra
```

## Preserved Registers

- preserved (saved) - `$s0` to `$s7`, `$ra` and `$sp`
  - the stack **above** the `$sp`
  - for **local varaibles** → optimization: check if the local varaible ever changed
- non-preserved (temporary) - `$t0` to `$t9`, `$a0` to `$a3` and `$v0` to `$v1`
  - the stack **below** the `$sp`
  - for **intermediate results**

```
diffofsums:
    #PROLOGUE
    addi $sp, $sp, -4
    sw $s0, 0($sp)
    #BODY
    add $t0, $a0, $a1
    add $t1, $a2, $a3
    sub $s0, $t0, $t1
    add $v0, $s0, $0
    #EPILOGUE
    lw $s0, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

# L18 The MIPS Architecture

## Compiling and Assembling

### compiler

- translates high-level code into assembly code for a target architecture
- needs to know
  - how to parse the high-level language
  - low-level specifics of the target architecture
  - how to map the high-level code to an architecture's assembly code

### assembler

- translates assembly code into machine code
  - used by the compiler
  - used by assembly programmers
- how does help
  - assembler directives

- calculates immediate values based on labels
- pseudo-instructions

**pseudo-instruction**

- two or more "real" MIPS instructions
- MIPS defines the register `$at` as the "assembler temporary" that only the assembler can use
- `la` - load a specific address into a register
  - translated to two instructions → `lui` and `ori`

```
.data
some_data:
    .word 0x123456

.text
    la $t0, some_data
    lw $t1, 0($t0)
```

**linker**

- combines one or more object files into a single executable file
- needs to know
  - where all the object files are
  - modern languages come with a package manager
  - for older languages

**executable file**

- header - how to read the binary data
- text - **instructions and the addresses** where they should be stored
- data - **global data and the addresses** where they should be stored

| Executable file header | Text Size | Data Size |
|---|---|---|
| | 0x34 (52 bytes) | 0xC (12 bytes) |
| **Text segment** | **Address** | **Instruction** |
| | 0x00400000 | 0x23BDFFFC |
| | 0x00400004 | 0xAFBF0000 |
| | 0x00400008 | 0x20040002 |
| | 0x0040000C | 0xAF848000 |
| | 0x00400010 | 0x20050003 |
| | 0x00400014 | 0xAF858004 |
| | 0x00400018 | 0x0C10000B |
| | 0x0040001C | 0xAF828008 |
| | 0x00400020 | 0x8FBF0000 |
| | 0x00400024 | 0x23BD0004 |
| | 0x00400028 | 0x03E00008 |
| | 0x0040002C | 0x00851020 |
| | 0x00400030 | 0x03E0008 |
| **Data segment** | **Address** | **Data** |
| | 0x10000000 | f |
| | 0x10000004 | g |
| | 0x10000008 | y |

```
addi $sp, $sp, –4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, –4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

### Loading

# L19 & 20 Micro-architecture

## terminology

| term | definition/description |
|---|---|
| datapath | operates on words of data |
| control unit | control the operation of the datapath |
| single-cycle | |
| muti-cycle | |
| pipelined | |

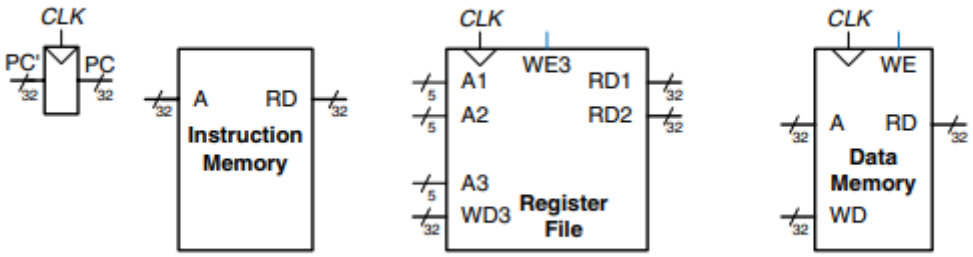



Figure 7.1 State elements of MIPS processor

# single-cycle datapath

### fetching instructions

- program counter - contains the address of the current instruction
- instruction memory - contains the data for the text segment

### sequential execution

- the common case (no branches) - PC = PC+4
- instructions are 32 bits
- instruction addresses are word-aligned

### implemnting `lw`

- `lw` is an I-type instruction
  - **rs** - 25-21 (base address)
  - **rt** - 20 - 16 (destination)
  - **imm** - 15 - 0 (offset)
  - effective address - the sum of the offset and base address

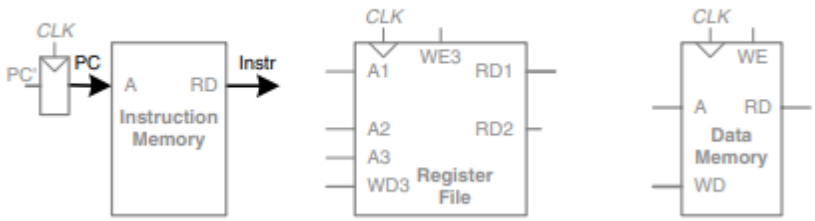



Figure 7.2 Fetch instruction from memory

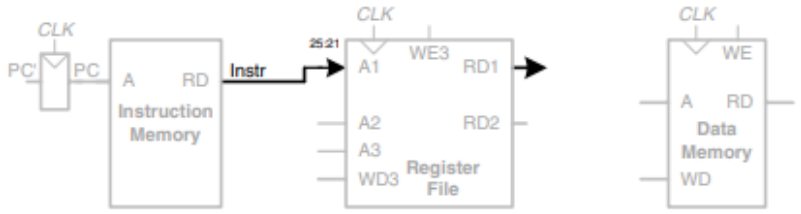



**Figure 7.3** Read source operand from register file

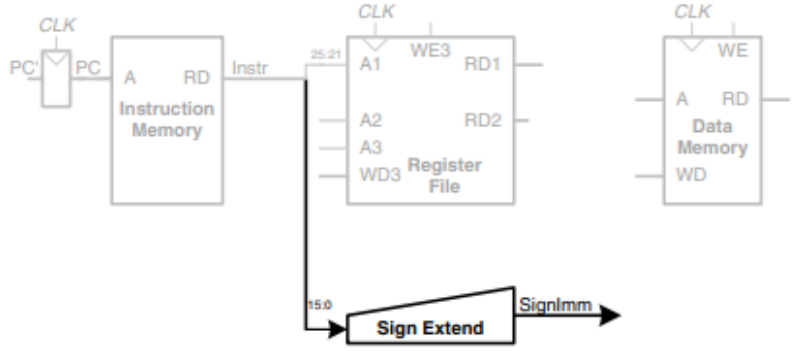



**Figure 7.4** Sign-extend the immediate

- ALUControl signal should be set to **010** to **add** the base address and offset

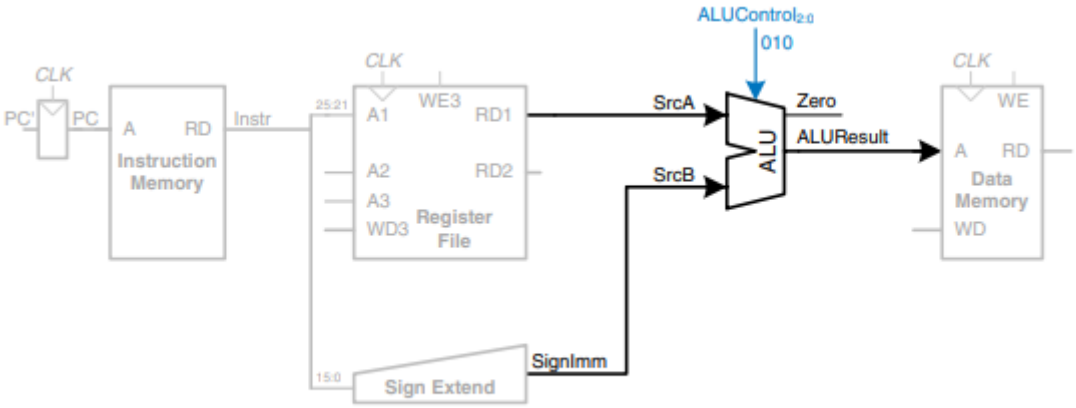



**Figure 7.5** Compute memory address

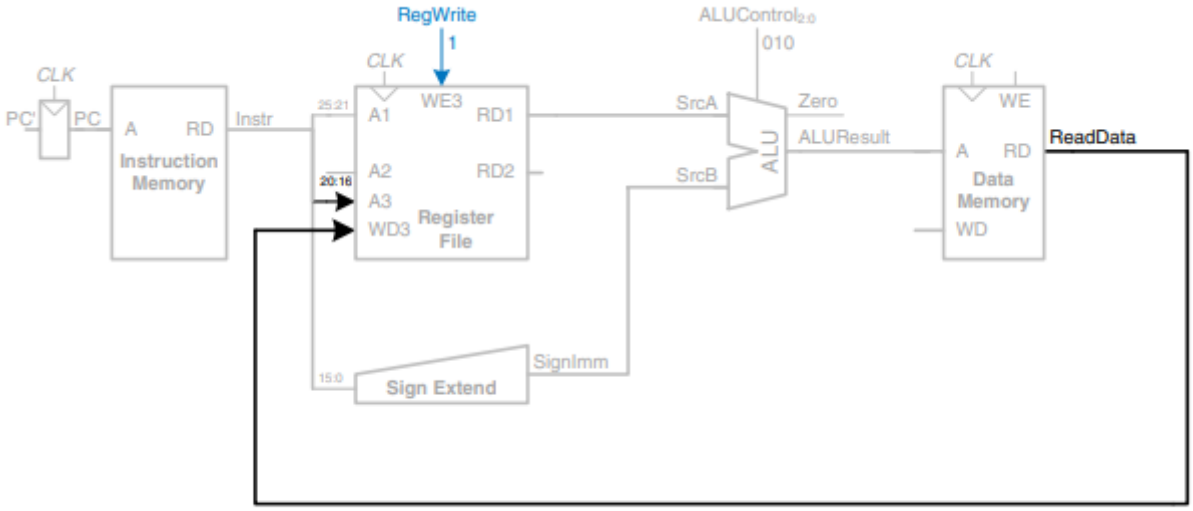



**Figure 7.6** Write data back to register file

- compute the address of the next instruction

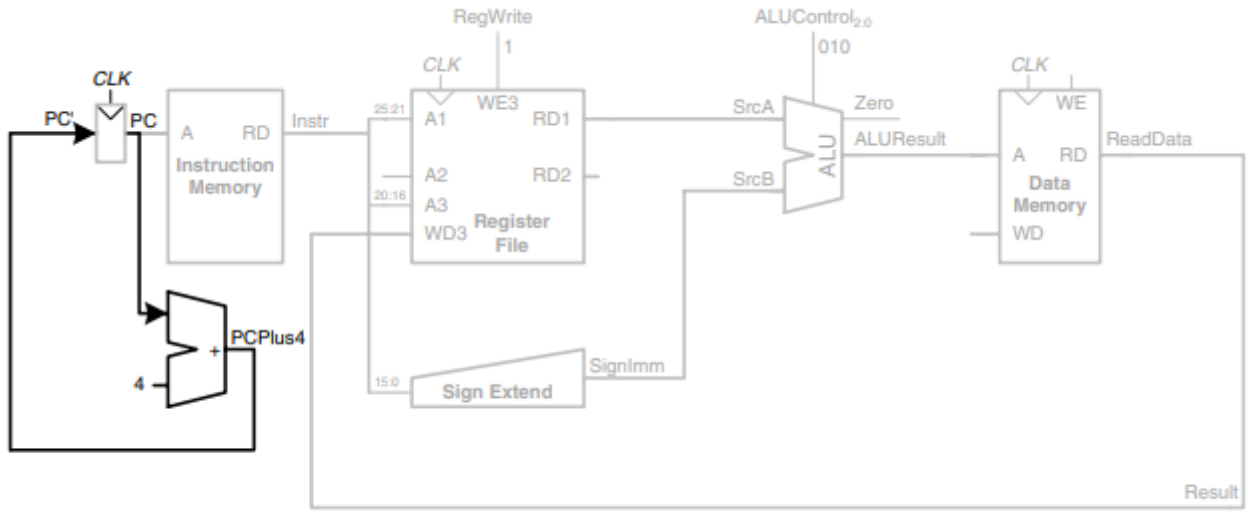



**Figure 7.7 Determine address of next instruction for PC**

### implementing `sw`

- different with `lw`
  - do not mutate a register, but mutates a value at an address in data memory

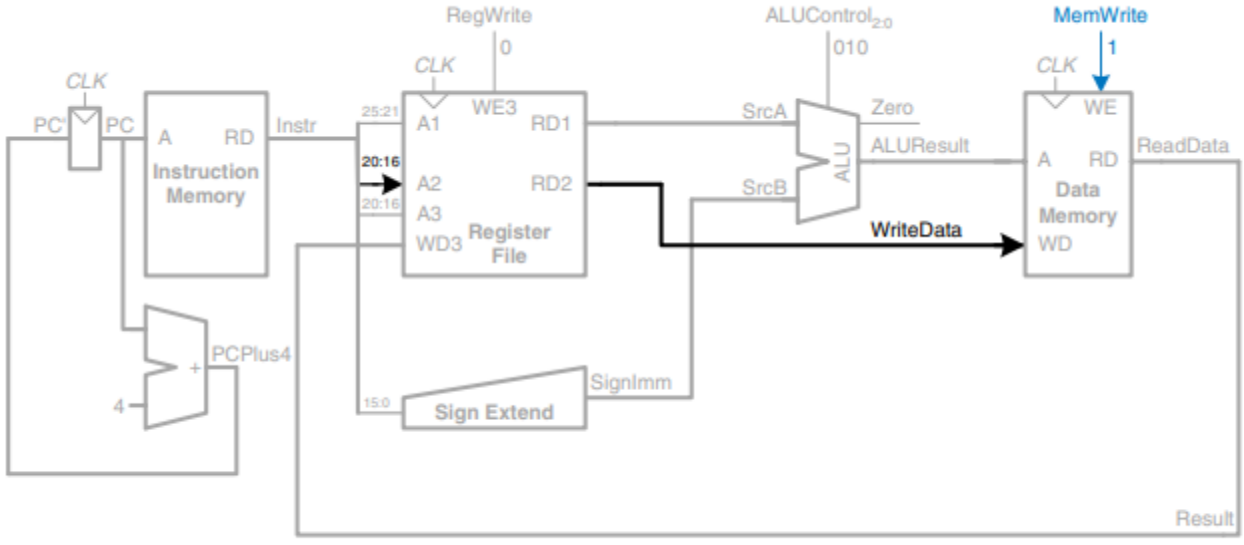



**Figure 7.8 Write data to memory for `sw` instruction**

**for R-type**

- different format
  - **rs** - 25:21 (source 1)
  - **rt** - 20:16 (source 2)
  - **rd** - 15:11 (destination)

## From before

- $ALUControl_{2:0}$
  - Depends on the R-type instruction
- RegWrite
  - Always!
- MemWrite
  - Never!

## New control signals

- RegDst
  - Is the destination register rd or rt?
  - i.e., I-type vs. R-type
- MemToReg
  - Is the data written to the register file from the ALU or memory?
- ALUSrc
  - Use immediate (I-type) or register (R-type) as second input to ALU?

- ALUSrc - 0 for R-type, 1 for `lw` and `sw` to choose SignImm
- RegDst - 0 for I-type, 1 for R-type

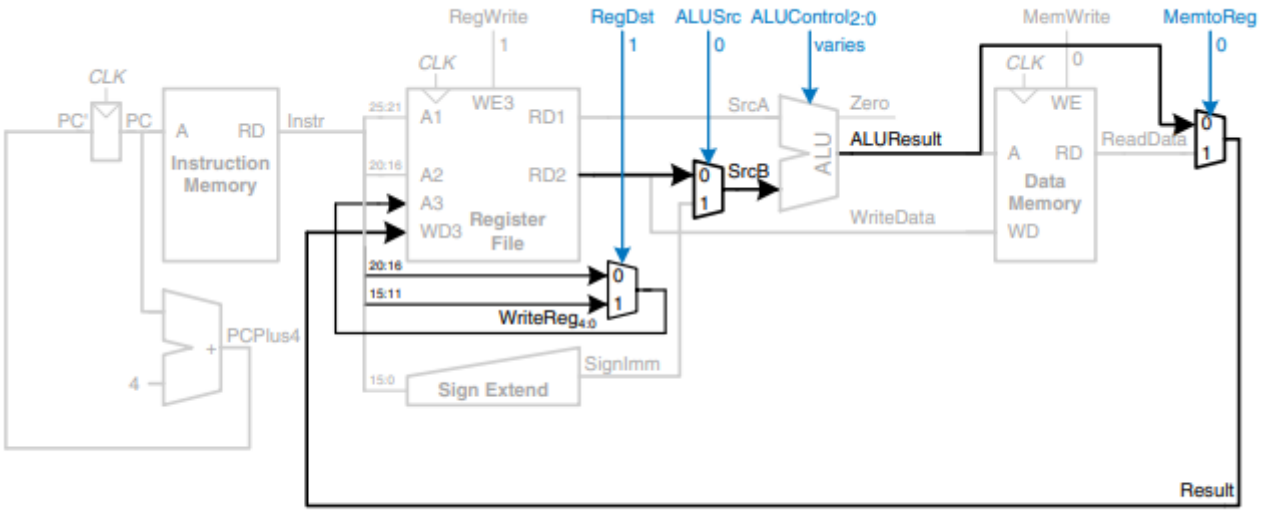



**Figure 7.9 Datapath enhancements for R-type instruction**

### for beq branch

- offset indicates the number of instructions to branch past
- $PC' = PC + 4 + SignImm \times 4$
- ZERO - some component to indicate whether the result is zero

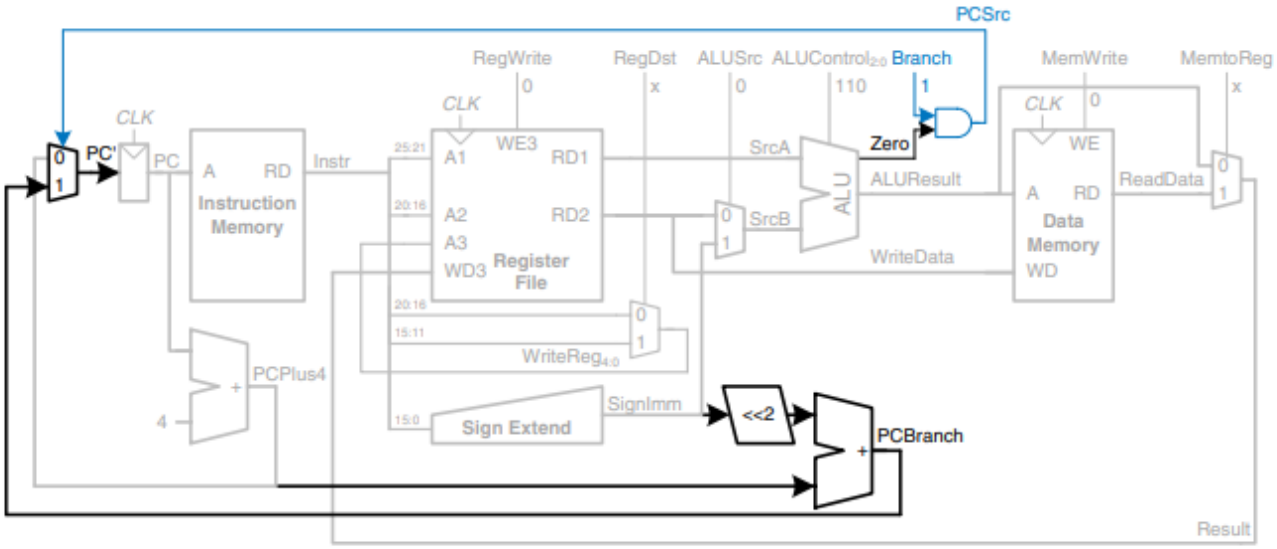
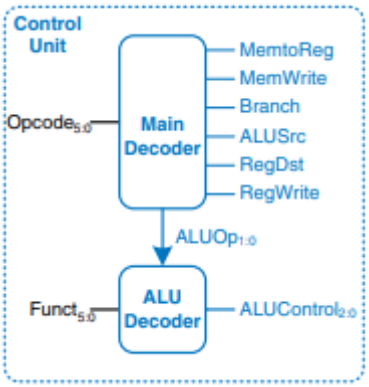



**Figure 7.10 Datapath enhancements for beq instruction**

## single-cycle control

### control unit

- funct - **R-type only**



**ALU decoder**

# "Factoring out" the ALU decoder

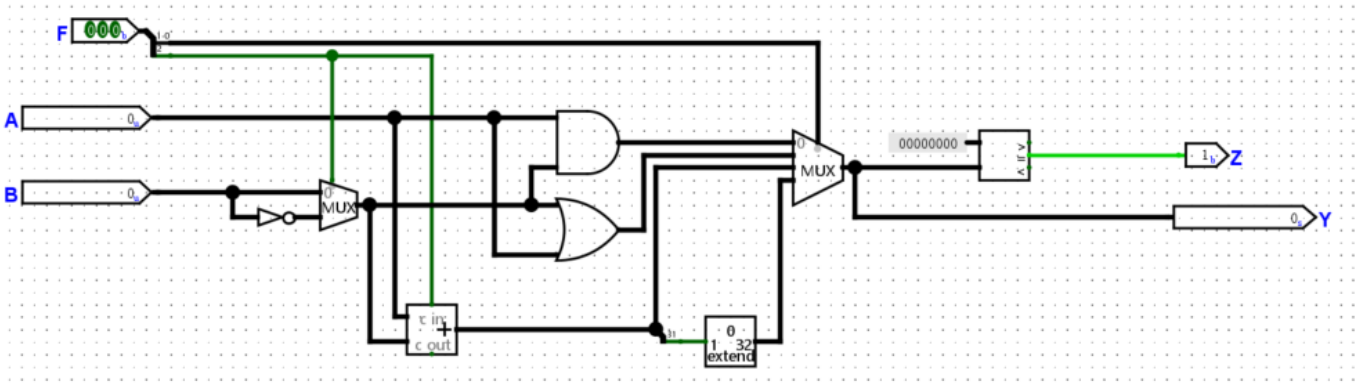
## Main Decoder

- Input: only the opcode
- Output
  - All control signals except $ALUControl_{2:0}$
  - New: $ALUOp_{1:0}$

## ALU Decoder

- Inputs
  - Funct
  - $ALUOp_{1:0}$
- Output
  - $ALUControl_{2:0}$

- supported operations



| ALUOp | Description |
|---|---|
| 00 | add |
| 01 | subtract |
| 10 | See funct |
| 11 | Not used |

| F | Operation |
|---|---|
| 000 | A and B |
| 001 | A or B |
| 010 | A + B |
| 011 | Unused |
| 100 | A and (not B) |
| 101 | A or (not B) |
| 110 | A - B |
| 111 | set less than |

**Table 7.2 ALU decoder truth table**

| ALUOp | Funct | ALUControl |
|---|---|---|
| 00 | X | 010 (add) |
| X1 | X | 110 (subtract) |
| 1X | 100000 (add) | 010 (add) |
| 1X | 100010 (sub) | 110 (subtract) |
| 1X | 100100 (and) | 000 (and) |
| 1X | 100101 (or) | 001 (or) |
| 1X | 101010 (slt) | 111 (set less than) |

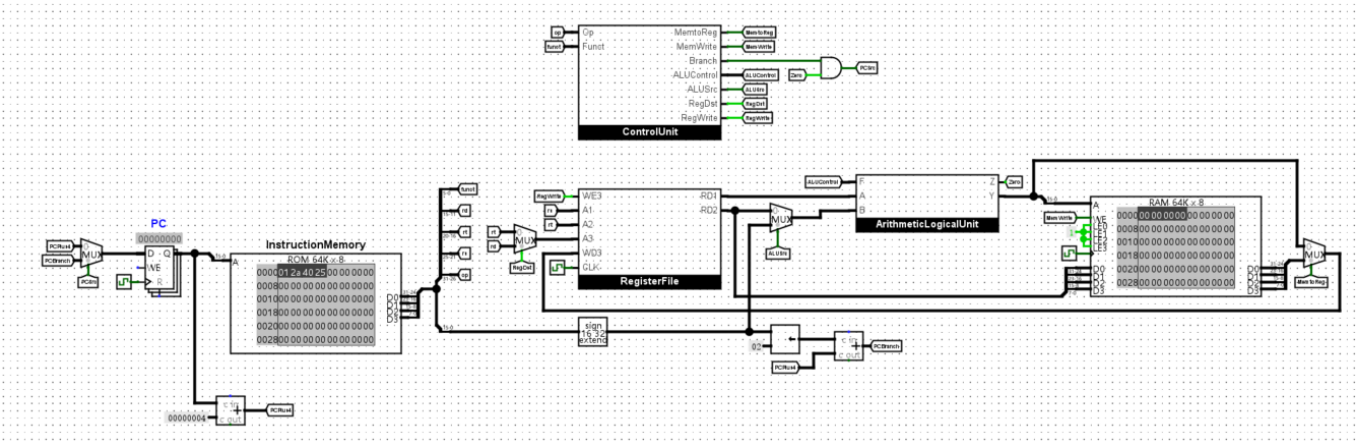
- $ALUControl_0 = ALUOp_1 \cdot Funct_0 + ALUOp_1 \cdot Funct_3$

**main decoder**

**Table 7.3 Main decoder truth table**

| Instruction | Opcode | RegWrite | RegDst | ALUSrc | Branch | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |

- $branch = Opcode_2$




# L21 Exceptions

## types

### by hardware

- by an input/output device → **interrupt**
- hardware malfunction

### by software

- divsion by zero
- bad memory accesses
- debugger brakpoints
- system calls → **traps**

**procedure**

> What happens when an exception occurs?

1. Stores the "cause" of the exception and the PC
2. Jumps to the exception handler procedure
3. Exception handler examines the cause and responds
4. Returns to the program that was running before the exception happened

**implement**

- exception handler's start address is always **0×80000180**
- new 32-bit registers
  - cause - stores the **reason** for the exception
  - exception PC - stores the **value** of the PC at the times of the exception

| Exception | Cause Register | |
|---|---|---|
| Hardware interrupt | 0x00 | 0000 0000 |
| System call | 0x20 | 0010 0000 |
| Breakpoint or DivideBy0 | 0x24 | 0010 0100 |
| Undefined instruction | 0x28 | 0010 1000 |
| Arithmetic overflow | 0x30 | 0011 0000 |

- cause and EPC are **not** in the register file
- `mfc $t0, Casue` → copy Cause into $t0

```
# PROLOGUE
# … save registers on stack

# BODY
mfc0 $t0, Cause
# if Cause == … respond …

# EPILOGUE
# … restore registers from stack
mfc0 $k0, EPC
jr $k0
```

**supporting**

- wite the value of the "next" PC to EPC
- EPCWrite → enable
- IntCause → select the right value
- CauseWrite → write enable
- we **cannot** do it with a J-type instruction

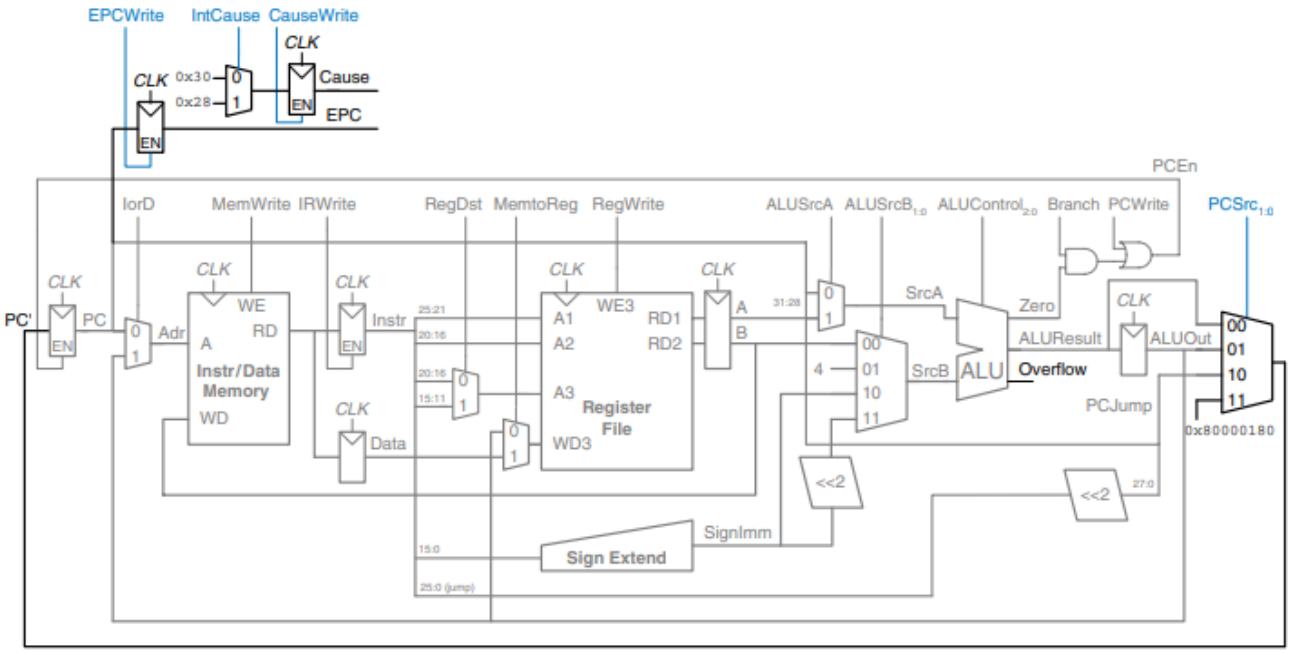
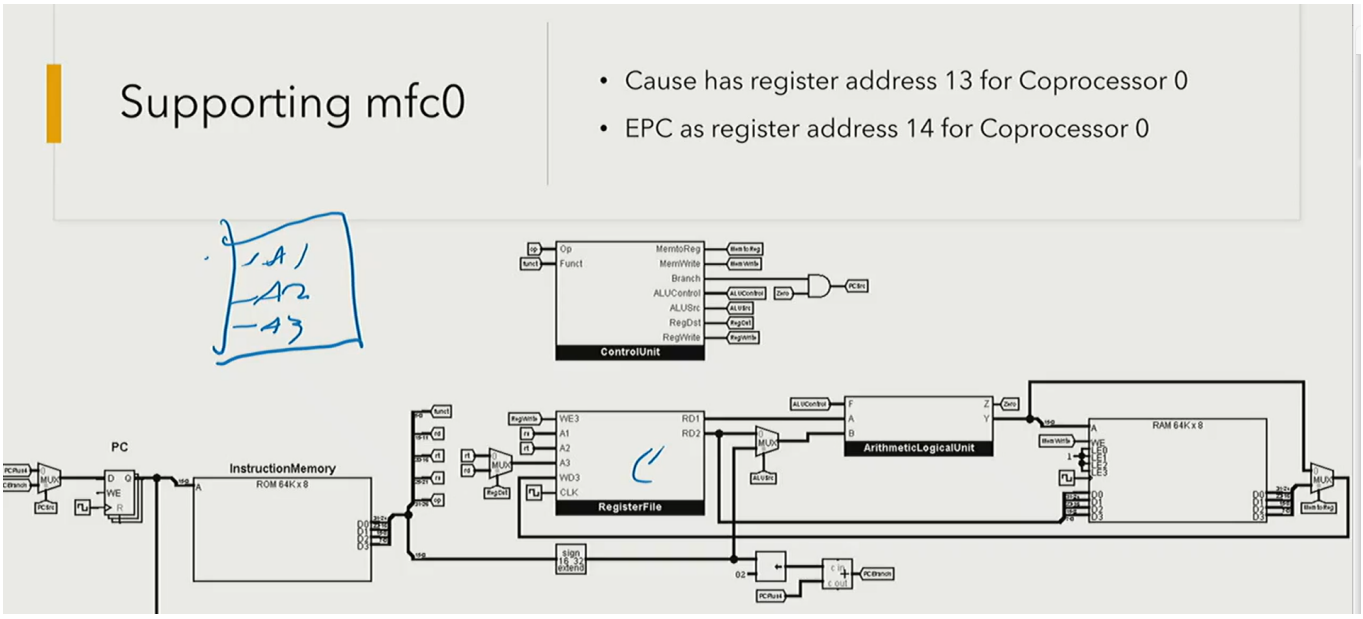



**Figure 7.62 Datapath supporting overflow and undefined instruction exceptions**




# L22 Performance Analysis

## evaluating a processor

### Area

- newer technologies allow more transistors to be packed on a single chip
- only some transistors are used for the processor
- in general, **more transistors** means a **larger processor**

### performance

- how long (execution time) does it take to execute a program
- a collection of "interesting programs" are **benchmarks**
- benchmarks are run on processors to evaluate their performance

### iron law of performance

$$Execution\ Time = \left(\#\ instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

- #instruction
  - influenced by **algorithms, the programmer, and the compiler**

- CISC → some instructions do more work than other

- cycles per instruction (CPI)
  - heavily influenced by the microarchitecture
  - a measure of latency
- seconds per cycle ($T_c$)
  - the critical path through one micro-architecture is **not the same** as another

## Microarchitecture trade-offs

**Goals**

- Minimize execution times
- More recently: minimize power consumption
- Even more recently: security

**Constraints**

- Area budget (e.g., number of transistors)
- Money (target cost of processor)
- Time (i.e., design time)

## evalutating MIPS microarchitecture

### single-cycle

- one instruction takes one cycle
- cycle time limited to slowest instruction

$$T_c = t_{pcq\_\mathrm{PC}} + t_{\mathrm{mem}} + \max[t_{RF\mathrm{read}}, t_{sext}] + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{\mathrm{mem}} + t_{\mathrm{mux}} + t_{RF\mathrm{setup}}$$

**Table 7.6 Delays of circuit elements**

| Element | Parameter | Delay (ps) |
|---|---|---|
| register clk-to-Q | $t_{pcq}$ | 30 |
| register setup | $t_{setup}$ | 20 |
| multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| memory read | $t_{mem}$ | 250 |
| register file read | $t_{RFread}$ | 150 |
| register file setup | $t_{RFsetup}$ | 20 |

**multi-cycle**

- one instruction takes multi cycles
- re-uses datapath better than single-cycle, but requires **intermediate** registers
- $T_c$ is **shorter** than single-cycle processor
- doing less work on each cycle
- CPI depends on the program

| Instructions | Number of Cycles |
|---|---|
| beq, j | 3 |
| sw, addi, R-type | 4 |
| lw | 5 |

**Example 7.7 MULTICYCLE PROCESSOR CPI**

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.[3] Determine the average CPI for this benchmark.

**Solution:** The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used. For this benchmark, Average CPI = (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12. This is better than the worst-case CPI of 5, which would be required if all instructions took the same time.

- B - billion,

**Single-cycle**

- Let # instructions = 1B
- Recall
  - $T_C$: 950 ps
  - CPI: 1
- Total execution time?

$(1B)(1)(950p) = 95s$

**Multi-cycle**

- Let # instructions = 1B
- Assume $T_C$: 325 ps
- Recall CPI: 4.12

$(1B)(4.12)(325p) \cong 134s$

**pipelined**

- executes several instructions simultaneously, but requires intermediate registers
- requires **additional logic** for dependencies between instructions

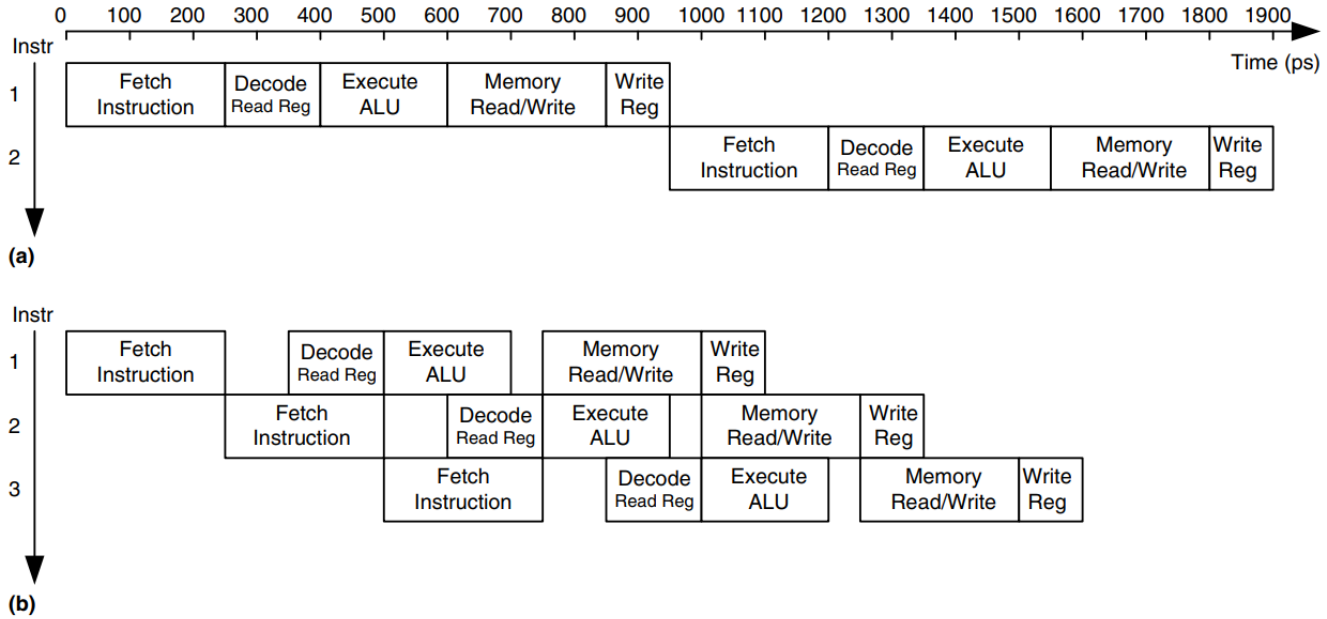



**Figure 7.43** Timing diagrams: (a) single-cycle processor, (b) pipelined processor

## Pipelined CPI

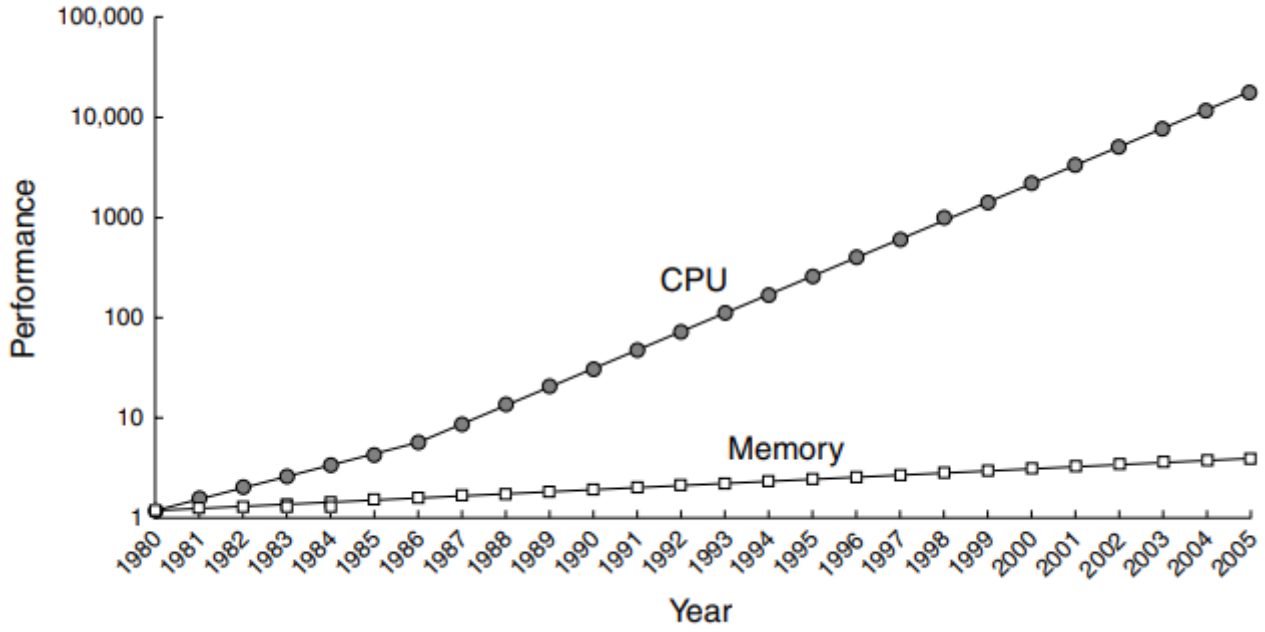
- Ideal CPI is 1
- But ideal is "never" reached due to
  - Dependencies between instructions
  - Branch instructions
- So, CPI = 1 + (overhead)

- Load-to-use Example
  - lw $t0, 0($s0)
  - addi $t1, $t0, 7

# L23 Memory Systems

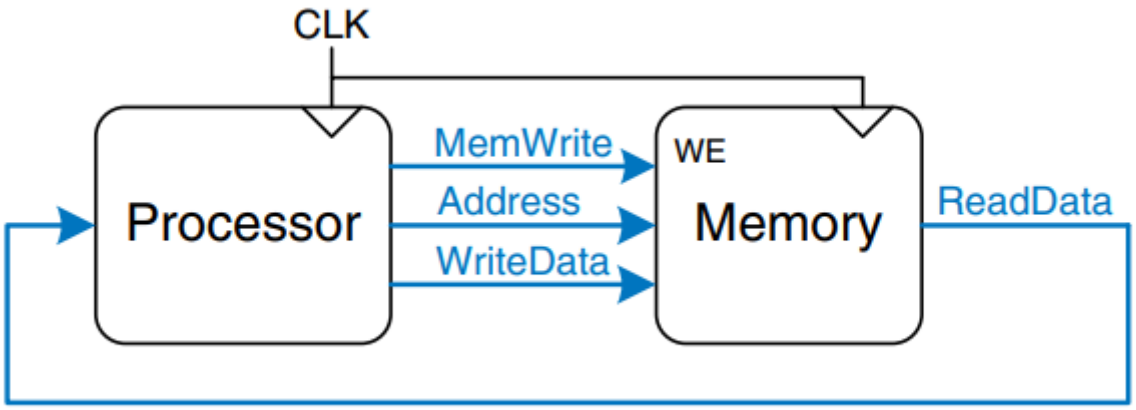
## motivation

- computer system performance depends on
  - microarchitecture
  - memory system
- the performance of memory systems has not kept pace with processors



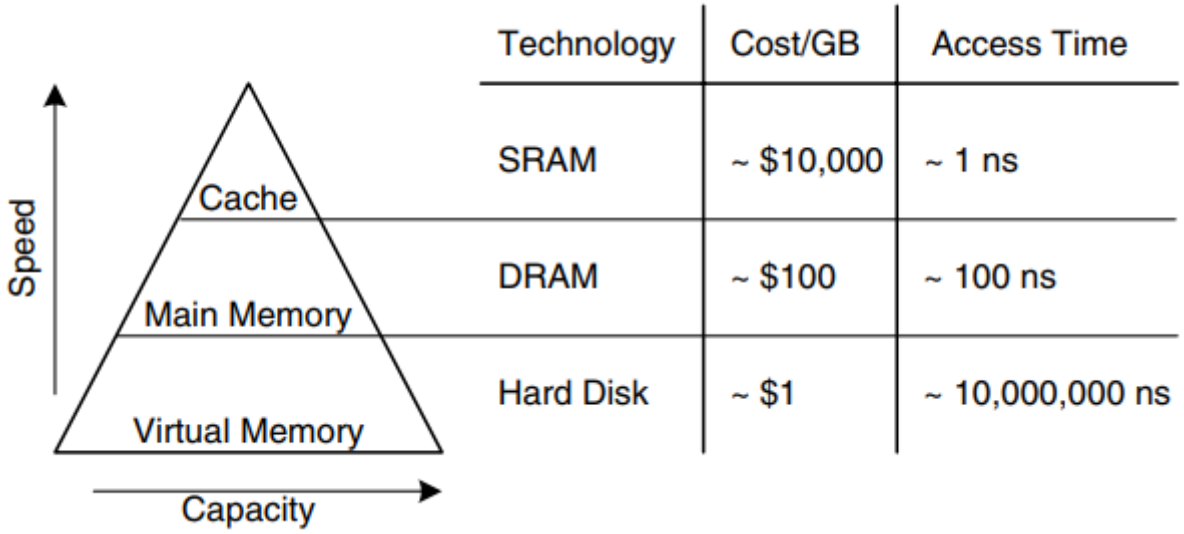

## memory interface

- DRAM access times is orders of **magnitude slower than our processor**




## trade-offs between memory

- access time: **lower is better**



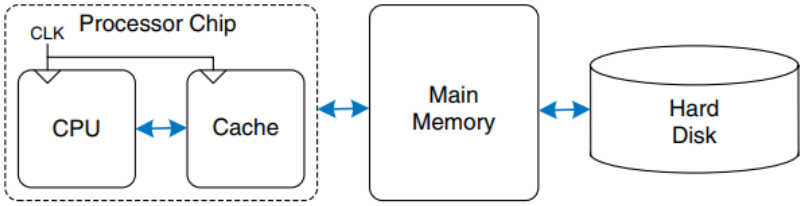




**Figure 8.3** **A typical memory hierarchy**

## memory system performance analysis

- limited capacity compared to main memory
  - significantly more than our register file
- stores both instructions and data
- data
  - in cache → **hit**, can return to processor quickly
  - **not** in cache → **miss**, need to **retrieve** the data from main memory
    - processor wait → **stall**
    - memory system performance impacts CPI

$$Miss\ Rate = \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate$$

$$Hit\ Rate = \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate$$

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

**Solution:** The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

**average memory access time(AMAT)**

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM})$$

---

**Example 8.2** CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates given in Table 8.1?

**Solution:** The average memory access time is 1 + 0.1(100) = 11 cycles.

---

**Table 8.1 Access times and miss rates**

| Memory Level | Access Time (Cycles) | Miss Rate |
|---|---|---|
| Cache | 1 | 10% |
| Main Memory | 100 | 0% |

**Example 8.3** IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

**Solution:** If the miss rate is $m$, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for $m$ requires a cache miss rate of 0.5%.

#### Amndahl's Law

the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance

## caches

#### capacity

- the number of words that a cache can hold
- a subset of main memory
  - on a miss, we retrieve data from memory

#### block

- has a block size $(b)$
- with capacity $C$ has $B = \frac{c}{b}$ blocks