CS2100: COMPUTER ORGANISATION
AY20/21 SEMESTER 1

ZHENG YONG ANG

Contents

Date: November 17, 2020.

1. Data Representation and Number Systems

We can convert from decimals to binary using either of the following methods:

(1) **Repeated division-by-2:** Successively dividing the decimal number by 2 until the quotient is 0. The remainders form the answer; the first remainder is the LSB and the last remainder is the MSB.
(2) **Sum-of-weight:** Determmine the set of binary weights whose sum is equal to the decimal number.

We can also convert from base-X to base-Y using the decimal system as a bridge.

Some of the common storage units/ data types include:

Unit/ Data type	Size
Bit	1 bit
Byte	8 bits
Word	4 or 8 bytes
char	1 byte
int	4 or 8 bytes
float	4 or 8 bytes
double	8 or 16 bytes

Representations for integers include:

(1) Sign-and-magnitude
   - MSB indicates the sign: 0 means positive, and 1 means negative.
(2) Complement
   (a) 1s complement
      - Negative values can be expressed as $2^n - X - 1$ (or bitwise complement)
   (b) 2s complement
      - Negative values can be expressed as $2^n - X$ (or bitwise complement + 1)
   - Benefit: more suitable for performing subtraction.
(3) Excess-K
   - K (bias/offset) will be added to the original value, before representing the result as binary.
   - Benefit: more suitable for comparing relative magnitudes of numbers.

Common format for floating point numbers: IEEE 754

- 32-bit platform: 1 sign bit, 8 exponent bits (excess-127), and 23 fraction bits (normalized to 1.X and take X only).
- 64-bit platform: 1 sign bit, 11 exponent bits (excess-1023), and 52 fraction bits.

2. C Programming

Pointers:

- Declaration: `int *ptr;`
- Usage: `ptr = &x;`
- Dereferencing: `*ptr = 0;` (same as setting $x = 0$)
- Pointer to another pointer: `int** pptr = &ptr;`

Arrays:

- Initialization: `int arr[3] = {1, 2, 3};`

Structures:

```
struct A { int a; double b; char c[3]; };
```

- Initialization: `struct A a = {'a', 'b', 'c'}` → `struct A a = {1, 2.0, {‘a’, ‘b’, ‘c’}};`
- Usage: `a.a = 10; a.c[0] = ‘z’;`
- Passing by address: `void hi(struct A *ptr) { ... }`
  - To access elements in the struct, we can either do `(*ptr).c[0];` or `ptr->c[0];`

3. MIPS

Each instruction is fixed-length 32-bits. Instruction formats include:

- R-format: `op $rd, $rs, $rt`
  - Instructions include: `add, sub, and, or, not, slt, srl, sll`

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

  - For `sll/srl`, `rs=0` and `rt` is the source register.
- I-format: `op $rt, $rs, Immd` or `op $rt, Immd($rs)`
  - Instructions include: `addi, andi, ori, slti, lw, sw, beq, bne`

`opcode`	`rs`	`rt`	`immediate`
6	5	5	16

  - The immediate is represented in 2s complement.
  - `beq/bne` branch relative to the program counter; i.e. `Immd=(target-(PC+4)>>2`
- J-format: `j Immd`

`opcode`	`immediate`
6	26

  - `j` does not branch relative to the program counter; i.e. `Immd=(`last 28 bits of target address`)>>2`
  - The first 4 bits of the target address needs to be the same as the first 4 bits of the program counter.

4. Processor

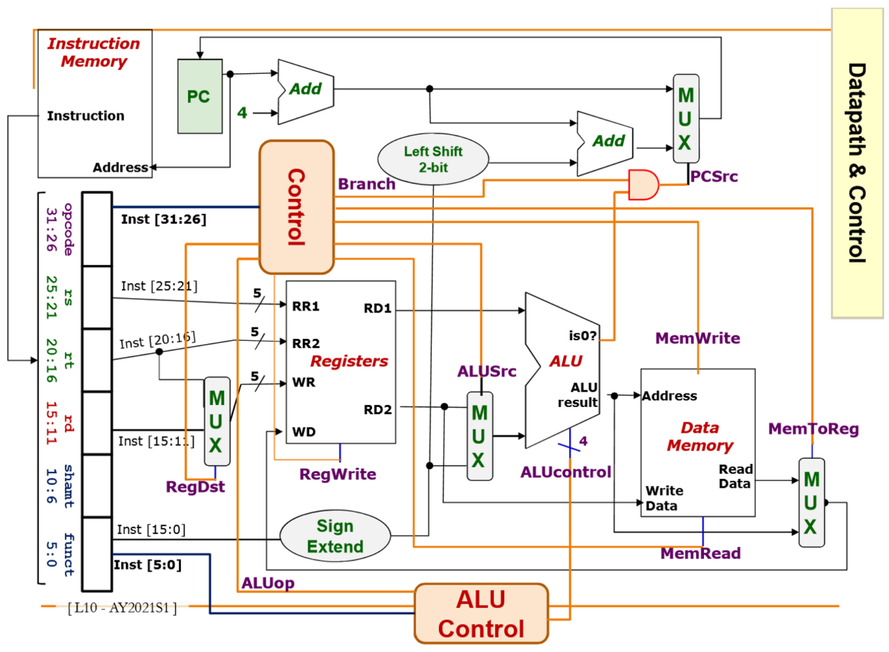


FIGURE 1. Complete datapath

Stages:

(1) Fetch
- Use PC to fetch the instruction from memory.
- Increment PC by 4 to get the address of the next instruction.

(2) Decode
- Read opcode to determine instruction type and field lengths.
- Read data from necessary registers.

(3) Execute

(4) Memory
- Only load and store instructions need to perform operations in this stage.

(5) Result write

Control signals:

Control Signal		Value
`RegDst`	Is `RegDst rd`?	0: write register = `rt`; 1: write register = `rd`
`RegWrite`	`RegWrite`?	0: no register write; 1: new value written
`ALUSrc`	Is `ALUSrc Immd`?	0: use RD2; 1: use SignExt(`Immd`)
`MemRead`	`MemRead`?	0: no memory read; 1: read memory with `Address`
`MemWrite`	`MemWrite`?	0: no memory write; 1: write RD2 into memory[`Address`]
`MemToReg`[1]	Is `Mem` value stored `ToReg`?	1: `WD = Read Data`; 0: `WD = ALU result`
`PCSrc`	`Branch && is0`	0: no branch; 1: branch taken

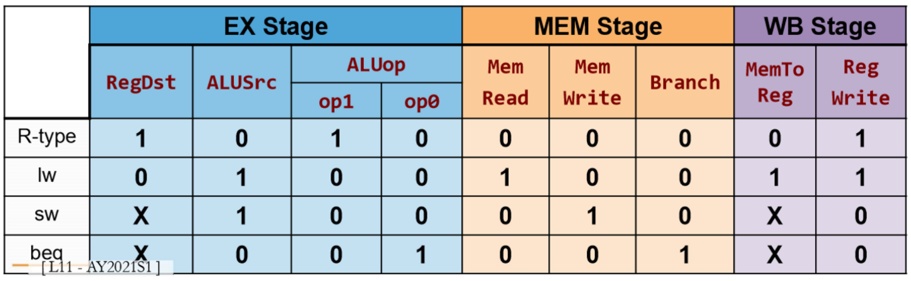
[1]Input of MUX is swapped in this case

`ALUControl`

`0000`	`AND`
`0001`	`OR`
`0010`	`add`
`0110`	`subtract`
`0111`	`slt`
`1100`	`NOR`

`ALUop`

`00`	`lw, sw`
`01`	`beq`
`10`	`add, sub, and, or, slt`



	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

[L11 - AY2021S1]

FIGURE 2. Control signals grouped according to pipeline stages

5. PIPELINING

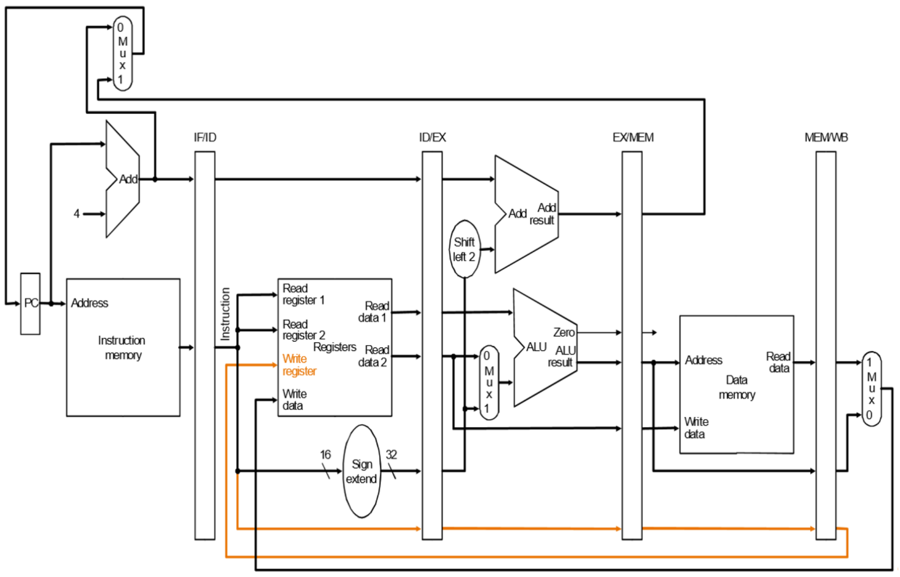


FIGURE 3. Datapath with pipeline registers

Pipeline registers pass down information in the following manner:

`curr` stage	`prev/curr` register supplies	`curr/next` register receives
`IF`	-	Instruction read from `Mem[PC]`, `PC + 4`
`ID`	2 read register numbers, 16-bit `Immd` sign-extended to 32-bit	Register values, 32-bit `Immd` value, `PC + 4`
`EX`	Register values, 32-bit `Immd` value, `PC + 4`	Branch target address, ALU result, `is0` signal, `RD2` value
`MEM`	Branch target address, ALU result, `is0` signal, `RD2` value	ALU result, memory read data
`WB`	ALU result, memory read data	Result written back to register file

Pipeline hazards include:

(1) **Structural hazards:** simultaneous use of a hardware resource.
- Solutions:
  (a) Pipeline stall
  (b) Split memory into data and instruction memory

(2) **Data hazards:** data dependencies between instructions.
- Read-After-Write (RAW)
- Solution:
  – Forward the result from one stage to another (bypass data read from register file)

(3) **Control hazards:** change in program flow.
- Solutions:
  (a) Early branch resolution: move branch decision calculation to earlier pipeline stage.
  (b) Branch prediction: guess the outcome before it is produced.
  (c) Delayed branching: execute other instructions while waiting for branch outcome.

6. Cache

Principle of locality: program accesses only a small portion of the memory address space within a small time interval.

Types of locality include:
- **Temporal locality:** if an item is referenced, it will tend to be referenced again soon.
- **Spatial locality:** if an item is referenced, nearby items will tend to be referenced soon.

Given a 32-bit memory address,
- The last N bits is the **offset**, where 2^N is the size of each cache block.
- The next M bits (from the right) is the **cache index**, where 2^M is the number of cache blocks in the cache.
- The remaining $32 - N - M$ bits form the **cache tag**.
- A **valid bit** indicates whether the data stored at that location is valid (not included in the 32-bits).

Tag	Index	Offset
$32 - N - M$	M	N

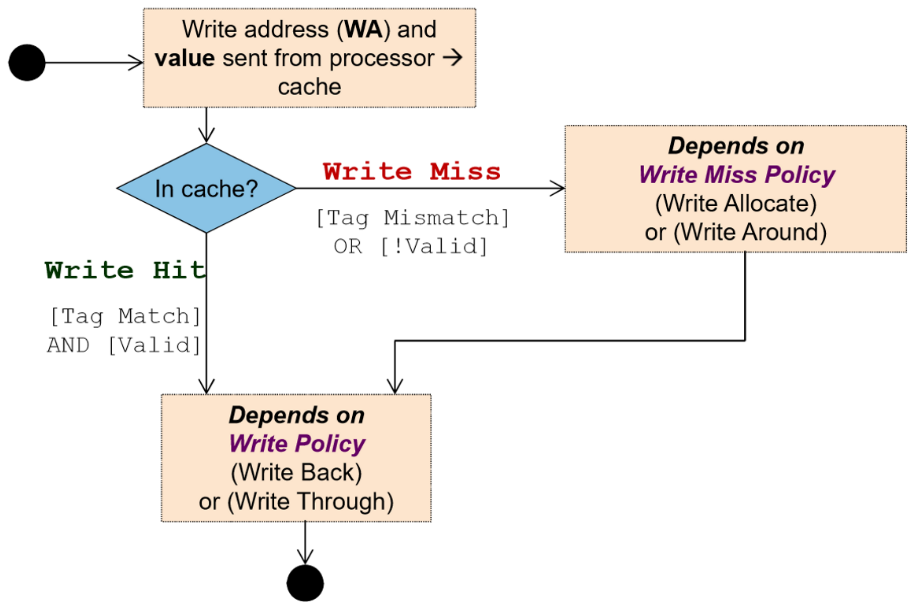

Write address (WA) and value sent from processor → cache
In cache?
Write Miss
[Tag Mismatch] OR [!Valid]
Depends on Write Miss Policy (Write Allocate) or (Write Around)
Write Hit
[Tag Match] AND [Valid]
Depends on Write Policy (Write Back) or (Write Through)

FIGURE 4. Cache store diagram

Cache write policies:

(1) **Write-through cache:** write data to both cache and main memory.
- Problem: write operates at the speed of main memory.
- Solution: place a write buffer between cache and main memory.

(2) **Write-back cache:** write data to cache, and only write to memory when cache block is replaced/evicted.
- Problem: wasteful if all evicted cache blocks are written back.
- Solution: add a *dirty bit* to each cache block to indicate that the cache block has been updated (sets dirty bit to 1).

Cache write miss policies:

(1) **Write allocation:** load the entire block into cache, change the required word in cache, and write to main memory depending on write policy.
(2) **Write around:** write directly to main memory; do not load the block to cache.

Cache misses:

(1) **Compulsory/cold miss:** first time a memory block is accessed.
- Solution:
  – Increase cache block size

(2) **Conflict miss:** two or more distinct memory blocks map to the same cache block.
- Solutions:
  (a) Increase cache size
  (b) Use set-associative caches

(3) **Capacity miss:** due to limited cache size.

Cache types:

(1) ***N*-way set associative cache:** cache which contains N cache blocks.
(2) **Direct-mapped cache:** basically 1-way set associative cache.
(3) **Fully associative cache:** memory block can be placed in any location in the cache (i.e. no cache index).
- Downside: need to search all cache blocks for memory access.

Block replacement policies:

(1) **Least recently used (LRU):** replace the block which has not been accessed for the longest time.
(2) **First In First Out (FIFO):** first block that enters cache will be replaced first.
(3) **FIFO with second chance:** uses a single bit to track whether the block has been accessed before, and replaces the block which has not been accessed. If all blocks have been accessed at least once, replacement is based on FIFO.
(4) **Random replacement (RR):** randomly selects a block to replace.
(5) **Least frequently used (LFU):** uses a counter to track the frequency that each block is accessed, and replaces the block that is least frequently accessed.

Cache framework:

	DM	SA	FA
Block placement	Only one block defined by index	Any one of the N blocks within the set defined by index	Any cache block
Block identification	Tag match with only one block	Tag match for all the blocks within the set	Tag match for all the blocks within the cache
Block replacement	No choice	Based on replacement policy	Based on replacement policy
Write strategy	Based on write and write miss policies	Based on write and write miss policies	Based on write and write miss policies

7. Performance

Definitions:

$$\begin{aligned}
\textit{Performance} &= \frac{1}{\textit{Response Time}} \\
\textit{Speedup} &= \frac{\textit{Performance}_x}{\textit{Performance}_y} \\
&= \frac{\textit{Response Time}_y}{\textit{Response Time}_x} \\
\textit{CPU Time} &= \frac{\textit{Time}}{\textit{Program}} \\
&= \frac{\textit{Cycles}}{\textit{Program}} \times \frac{\textit{Time}}{\textit{Cycles}} \\
&= \frac{\textit{Instructions}}{\textit{Program}} \times \frac{\textit{Cycles}}{\textit{Instructions}} \times \frac{\textit{Time}}{\textit{Cycles}} \\
\textit{CPI} &= \frac{\textit{CPU Time} \times \textit{Clock Rate}}{\textit{Instructions}} \\
&= \frac{\textit{Clock Cycles}}{\textit{Instructions}}
\end{aligned}$$

Amdahl's Law: performance is limited to the non-speedup portion of the program.

8. Logic Gates

Definitions:

- **Minterm:** a minterm of n variables is a *product term* that contains n literals from all the variables.
  - More simply, minterms are the entries of a truth table which return a value of 1.
- **Maxterm:** a maxterm of n variables is a *sum term* that contains n literals from all the variables.
  - More simply, maxterms are the entries of a truth table which return a value of 0.

- **Implicant:** a product term that could be used to cover several minterms of the function.
- **Prime implicant:** the largest possible implicant for a group of minterms.
- **Essential prime implicant:** the prime implicant that contains 1 or more unique minterms.

Simplification:

- K-map:

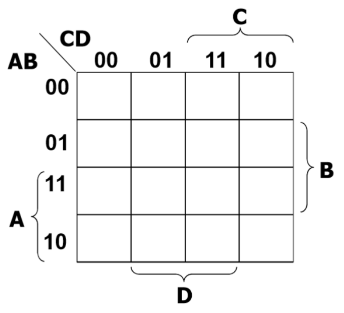


Figure 5. 4-variable K-map

- Algorithm:
  1. Draw the PIs for each minterm in the K-map.
  2. Using the PIs from step 1, take all the EPIs.
  3. Choose the smallest collection of PIs for the rest of the minterms that are still not covered.

9. Combinatorial Circuits

Basic components:

- **Half adder:** only adds two bits.

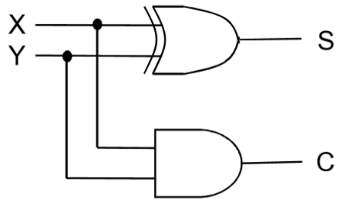


Figure 6. Half adder

- **Full adder:** adds two 1-bit binary numbers.

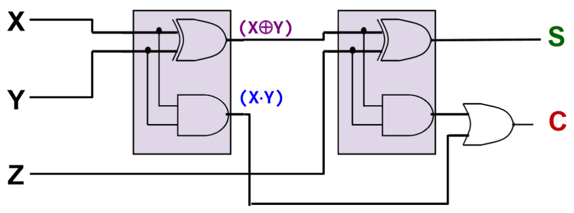


Figure 7. Full adder

- *N*-bit parallel/ ripple-carry adder: adds two *N*-bit binary numbers.
  - Can be constructed by cascading full adders.
- **Magnitude comparator:** compares two values *A* and *B* to check their relative ordering.

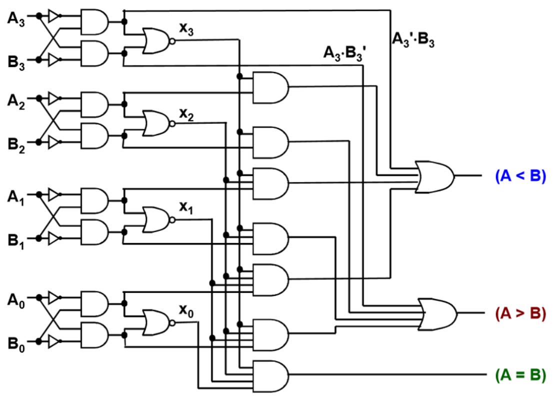


Figure 8. Magnitude comparator

Circuit delay: the output of a circuit with inputs taken at times $t_1, t_2, \dots, t_n$ and delay t will be obtained at time $\max\{t_1, t_2, \dots, t_n\} + t$.

Medium-Scale-Integration (MSI) components:

- **Decoder:** takes an *N*-bit code and converts it into one of its 2^N unique outputs.

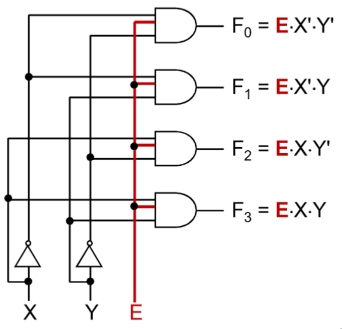


Figure 9. 2 × 4 decoder with enable

  - Larger decoders can be constructed in the same way as parallel adders.

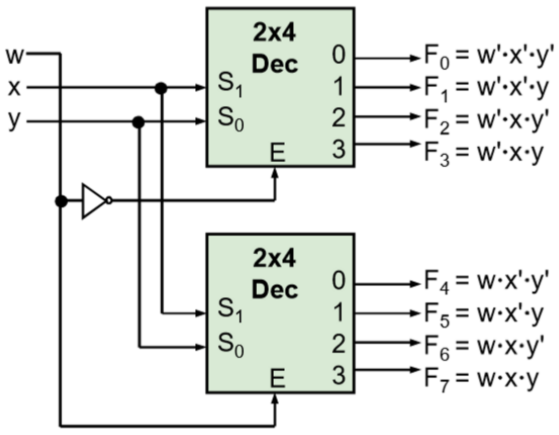


FIGURE 10. 3×8 decoder

- **Encoder:** performs the converse action of decoders.

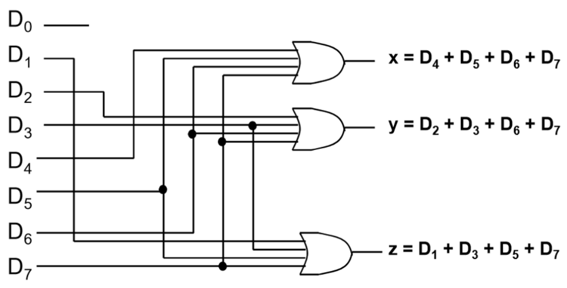


FIGURE 11. 3×8 encoder

  - This diagram can be constructed from a truth table.
- **Demultiplexer:** directs input data to the selected output line among 2^N possibilities (the circuit is identical to decoder with enable).

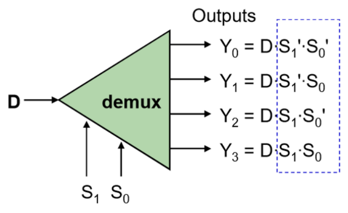


FIGURE 12. 1-to-4 demultiplexer

- **Multiplexer:** steers one of the 2^N inputs to the output line.

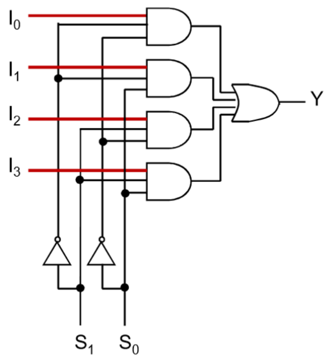
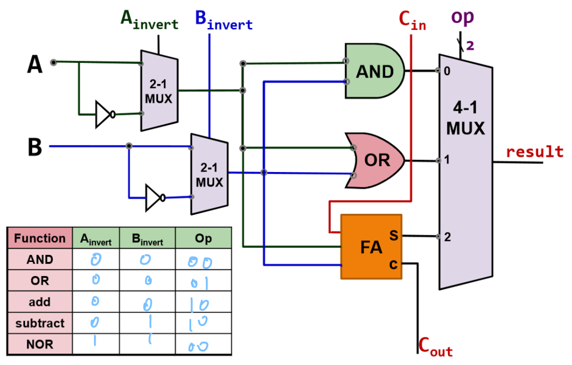


FIGURE 13. 4-to-1 multiplexer

- ALU implementation:



Function	A_{invert}	B_{invert}	Op
AND	0	0	00
OR	0	0	01
add	0	0	10
subtract	0	1	10
NOR	1	1	00

FIGURE 14. ALU implementation

10. SEQUENTIAL CIRCUITS

Memory elements: a device which can remember values indefinitely, or change values on command from its inputs.

- Characteristic table:

Command (at time t)	$Q(t)$	$Q(t+1)$
Set	$\times$	1
Reset	$\times$	0
Memorize/ No change	0	0
Memorize/ No change	1	1

- Clocking signal:

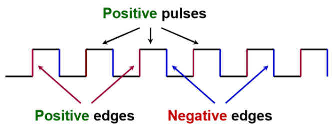


FIGURE 15. Clocking signal in memory elements

- **Latches:** pulse-triggered elements.
  (1) S-R Latch (Set-Reset Latch)
    - Active high:

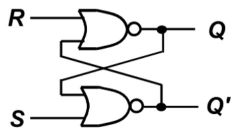


FIGURE 16. Active high S-R latch

S	R	$Q(t+1)$	Action
0	0	$Q(t)$	No change.
0	1	0	Latch **reset**.
1	0	1	Latch **set**.
1	1	$\times$	Invalid condition.

- Active low:

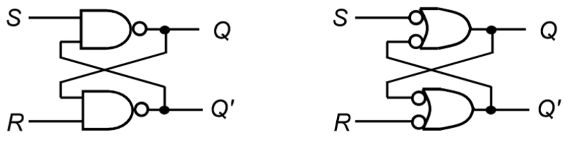


FIGURE 17. Active low S-R latches

S	R	$Q(t+1)$	Action
0	0	$\times$	Invalid command.
0	1	1	Latch **set**.
1	0	0	Latch **reset**.
1	1	$Q(t)$	No change.

- Gated: S-R latch with enable input, outputs changes only when EN is 1.

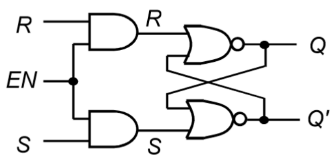


FIGURE 18. Gated S-R latch

(2) D Latch (Data Latch)

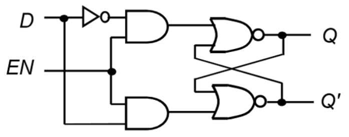


FIGURE 19. Gated D latch

EN	D	$Q(t+1)$	Action
1	0	0	Reset.
1	1	1	Set.
0	$\times$	$Q(t)$	No change.

- **Flip-flops:** edge-triggered elements.
  (1) S-R Flip-Flop

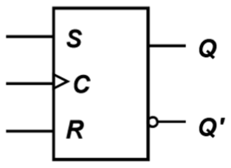


Figure 20. S-R flip-flop

S	R	CLK	$Q(t+1)$	Action
0	0	$\times$	$Q(t)$	No change.
0	1	$\uparrow$	0	Reset.
1	0	$\uparrow$	1	Set.
1	1	$\uparrow$	$\times$	Invalid condition.

  (2) D Flip-Flop

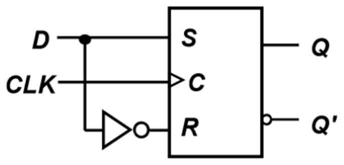


Figure 21. D flip-flop

D	CLK	$Q(t+1)$	Action
0	$\uparrow$	0	Reset.
1	$\uparrow$	1	Set.

  (3) J-K Flip-Flop

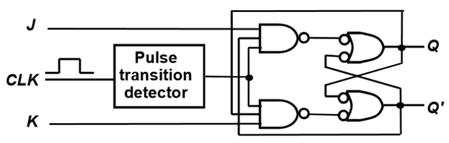


Figure 22. J-K flip-flop

J	K	CLK	$Q(t+1)$	Action
0	0	$\times$	$Q(t)$	No change.
0	1	$\uparrow$	0	Reset.
1	0	$\uparrow$	1	Set.
1	1	$\uparrow$	$Q(t)'$	Toggle.

(4) T Flip-Flop (Toggle Flip-Flop), basically D flip-flop without `NOT` gate

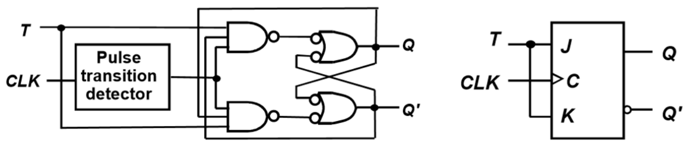
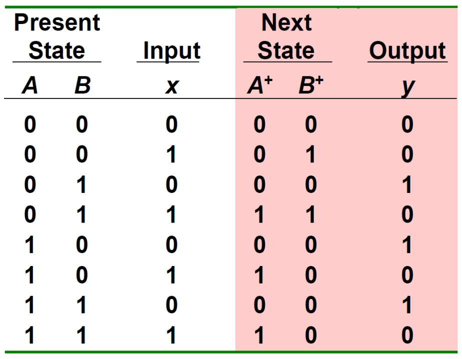


FIGURE 23. T flip-flop

T	CLK	$Q(t+1)$	Action
0	↑	$Q(t)$	No change.
1	↑	$Q(t)'$	Toggle.

Analysis:

- **State table:** consists of all possible binary combinations of present states and inputs (similar to truth tables).



Present State		Input	Next State		Output
A	B	x	A^+	B^+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

FIGURE 24. Sample state table

- **State diagram:** summarizes the behaviors between different states.

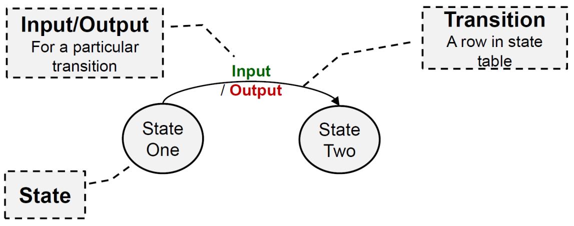


FIGURE 25. State diagram