**[Number System] 2’s Comp. conversion**

Positive Numbers

* Comp = original (补码=原码)
* Original = comp (原码=补码)

Negative Numbers

* **Comp = reverse(abs(original)) + 1 (补码=正数的反码+1)**
* **Original = -reverse(comp-1) (原码=补码-1/反转/再加负号)**

Negative Numbers

1. **统一加减法**：减法可以转换为加法来处理。

2. **简化溢出处理**：正负溢出方式相同，简化溢出的检测逻辑。

3. **避免两个零**：假若不用补码，则, 都代表0

**Overflow detection**

* Positive + Positive = Negative
* Negative + Negative = Positive

**Multiplication**

A diagram of a product

Description automatically generated

1. Left-shift multiplicand B
2. Right shift multiplier Q (so each time we can take Q[0])
3. A (product) = A + B \* Q[0]

**Floating Point Notation**

A close-up of a white rectangular object

Description automatically generated

* Significand: Normalized to start with **1 ()**
* **Bias = 2^(k-1)-1,** k is the number of bits of the “biased exponent”
* Bias Exponent:

Decimal Fraction -> Binary Float Point:

−29.21875 32-bit IEEE floating-point

A number lines and numbers

Description automatically generated with medium confidenceA math equations with numbers

Description automatically generated

Therefore

A math equations and numbers

Description automatically generated

**MIPS**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A white and black text on a white background

Description automatically generated

**A diagram of a stack

Description automatically generatedJ-type instructions**

* Only 26bits address
* [leftmost 4 bits of PC] + [26bit address] + [00, 2 zeros]

**I-type instructions**

* The immediate value “imm” only have 16 bits
* Zero extend leftwards to extend it to 32 bits

**R-type instructions**

* $rs = $rt + $rd. 5 Bit each register (since we have 32 in total)
* Shamt used in shift function

MIPS Instruction Cautions & Notes

**Bit-wise operations**

* ORI: Must be **non-negative** (2’s comp. meaningless after zero extend). ORI with zero can load **non-negative** constant to register
* ANDI: **sign irrelevant** (AND with zero padding always be 0). Good for clearing registers.
* XORI: **sign irrelevant**. Good for flipping specific bits of a register (by setting some bits of the immediate value to 1)
* NOR: equivalent to **NOT** (anything NOR 0, flips its bit)
* SLL: implement **No-Op** (sll $0, $0, 0)
* SRL: must be **non-negative** (due to zero padding on the left)
* SRA: **sign extend** left (extend with leftmost bit)

**Arithmetic operations**

* ADDU: **NOT unsigned**. It means **overflow is ignored**
* ADD: **overflow is detected**
* ADDIU: overflow ignore, **sign extended**
* Same for SUBU / SUB (overflow ignored / detected)
* MULTU: multiply unsigned (no overflow check)
* MULT: multiply signed (no overflow check)
* MFHI D: move ‘hi’ register -> ‘D’ register
* A table with numbers and letters

  Description automatically generatedMFLO D: move ‘lo’ register -> ‘D’ register
* DIVU: division for unsigned. **Remainder -> hi ; Quotient -> lo**

**Memory Access Operations**

* LW $1, imm($2): $1 <- RAM [ $2 + SignExtend(imm) ]
* SW $1, imm($2): RAM[ $2 + SignExtend(imm) ] <- $1
* LUI $1, imm: copy imm to upper 16bits of $1
* ORI $1, $0, imm: copy imm to lower 16bits of $1
* A table with numbers and letters

  Description automatically generated.data Directive: Declare variables & constants. Data section start at 0x10000000.
* .word Directive: Allocates a 4Bytes. (.data 69 / .data 1, 2, 3, 4)

**Control Flow Operations**

* J / JAL, PC + 4 before jump, thus need Branch Delay SLot

**Subroutine**

* $ra (return address) <- PC + 8
* $sp (stack pointer): points to stack top (smallest address)
* Stack push: subu $sp, $sp, 4 -> sw $t0, ($sp) --- ($sp) equiv 0($sp)
* Stack pop: lw $t0, ($sp) -> addu $sp, $sp, 4

**Boolean Algebra**

A screenshot of a math test

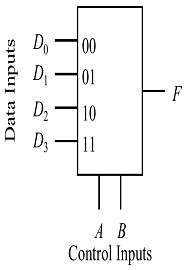
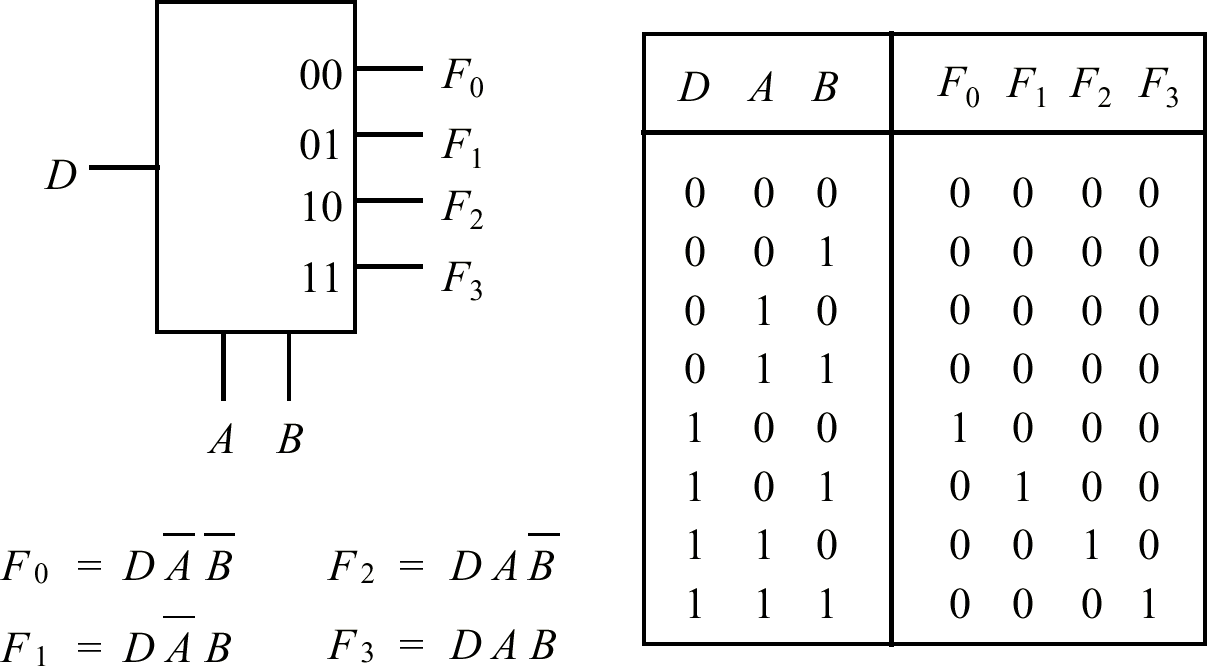
Description automatically generatedNumber of Possible Switching Functions: (N input variables)

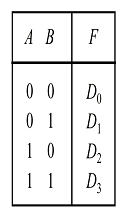
Boolean Function Duality **(Right Side of Above Table)**

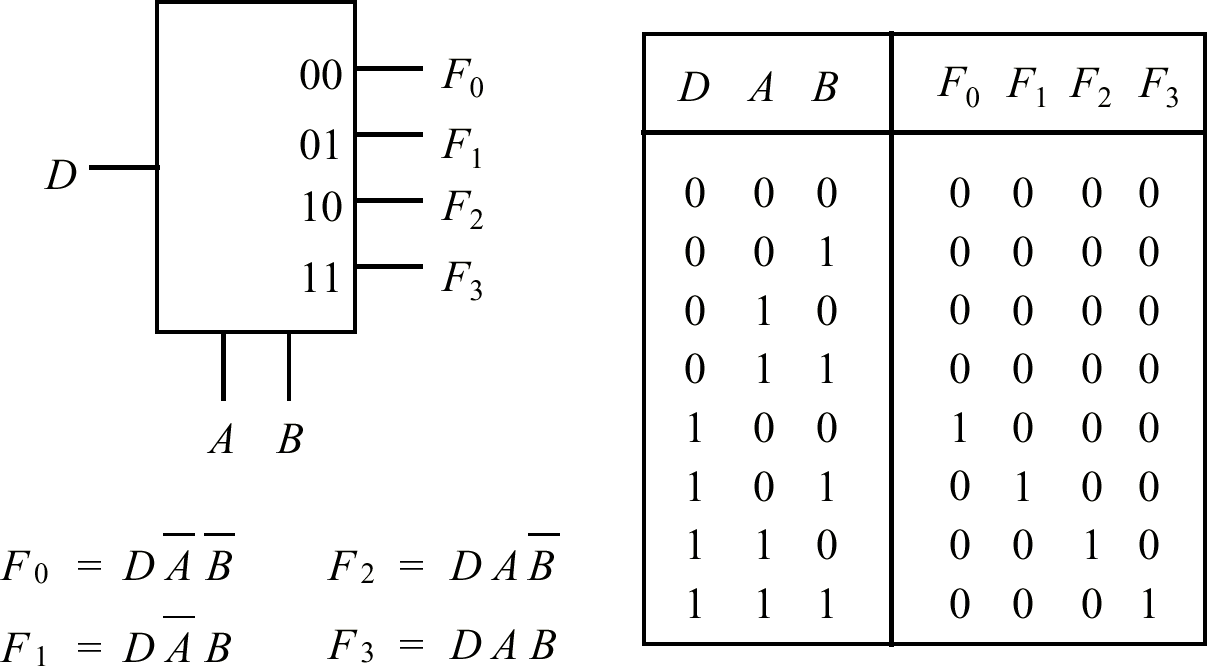
* Replace AND with OR ; Replace OR with AND
* Replace constant 1 with 0 ; Replace 0 with 1
* New function has exact opposite output.

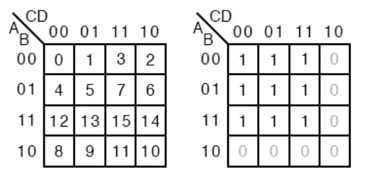
**Combinational Logic**

**Multiplexer (MUX) reduce wire usage Demultiplexer (DeMUX)**

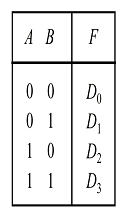








Truth Table



**Karnaugh Map (K-Map)**

* Bit order: 00 01 11 10 (both row and column)
* Group biggest blocks of 1s.
* “X” or “don’t care” don’t necessarily need to be circled. Only circle them when it simplifies things

**Minterm:** Product term in SOP form

**Maxterm:** Sum term in POS form

**Quine-McKluskey tables**

* Idea: if 2 minterms differ **only in one position**, can be simplified
* AB + AB’ = A(B + B’) = A

**Procedure**

1. Group minterms according to **the number of 1’s**
2. Check adjacent groups for simplification (group 1 & 2, 2 & 3 …)
3. Merge minterms only differ by one position. Denote the differing position with ‘-‘ (canceled)
4. Repeated terms can be cancelled
5. Repeat 1~5, Until Can’t merge. Now we get **prime implicant**: a minterm that cannot be further simplified with other minterms

A white grid with black and yellow dots

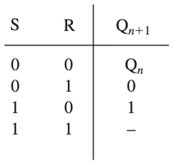
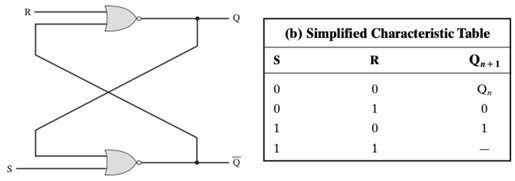
Description automatically generated**Prime Implicant Table:** each prime implicant covers some minterms

上面表格中如有minterm**仅被1个prime implicant 覆盖**，那么这个prime implicant是**essential prime implicant**

**Sequential Circuit**

* Not only dependent on current (comb. circuits) but also past behavior (i.e., storage element / memory)
* Output (next state) = current input + current state.
* Flip-flops: **bistable** device (two stable states – doesn’t change without external input). Outputs: complements each other.

**SR Flipflops**

****

**Edge Triggered (or Clocked) S-R Flip-Flop**

A diagram of a circuit

Description automatically generated

**D Flip-Flop**

A diagram of a circuit

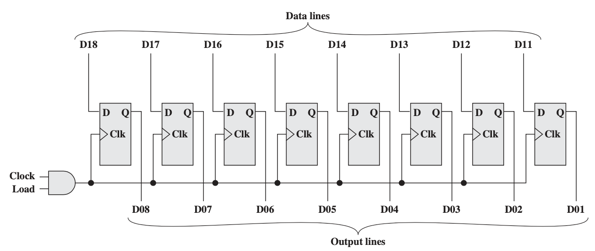
Description automatically generated

**JK Flip-Flop**

* Unlike SR Flip-Flop (which can’t take S=1, R=1 as input)
* JK Flip-Flop allows all combination of input (including 1 1)
* Output tick between 1, 0 each clock cycle when J=1 K=1

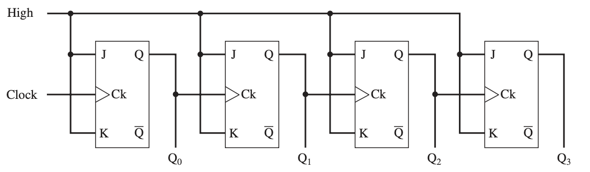
A diagram of a circuit

Description automatically generated

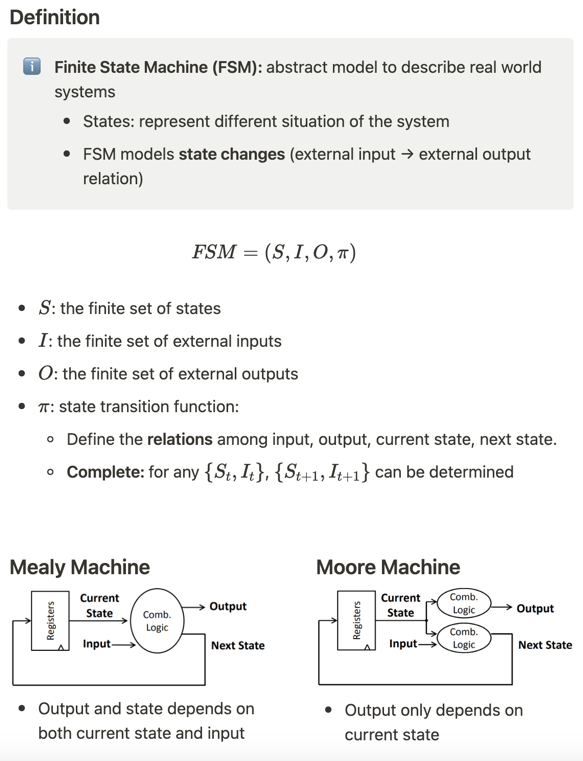
**8-bit Parallel Registers**

**Counter** (register whose value incr. by 1 every tick)

* For counter with flip flops, the value ranges from 0 ~
* Asynchronous counter: state of the FF will NOT change same time
* Synchronous counter: state of the FF changes at the same time

**Ripple Counter**

**Finite State Machine**



* **Registers = Memory Component**
* **Comb. logic = State transition Function**

A table with numbers and letters

Description automatically generated**FSM Model -> Circuit**

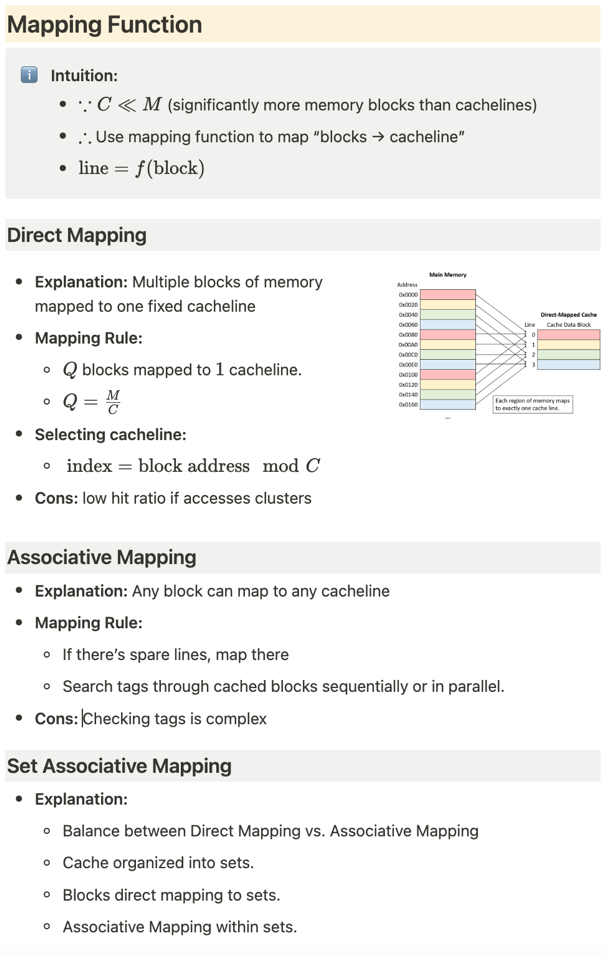
* Model the problem using FSM
* Construct **excitation table** of the circuit

**Columns:**

* : current state
* : Input
* : next state
* Input to Flip-Flop

Basically:

* FSM -> State table
* FF -> FF ExTable
* **Circuit table =** FF ExTable + State Table
* K-map simplification: All pairs of **(State Q) vs (Flip-Flop Input)**

**Cache Mapping**

**Cache Performance**

* Miss Rate = 1 – Hit Rate *( % of mem accesses not found in cache)*
* Hit Time = Time to deliver word from cache -> memory
* Miss Penalty = extra time needed (to access memory) due to miss

**A close-up of a sign

Description automatically generatedMemory Address**

* **Block Offset:** index of each word within the block.
* **Index:** indicates which cacheline
* **Tag:** unique identifier of blocks mapped to same cacheline (blocks of same cacheline have ***different tags***)

**Calculations:**

* bit address ->  **addressable words**
* words per block ->  **blocks**
* **Offset**: bits --- **log (Block Size) bits**
* **Index:** bits --- **log (Num. of Cacheline) bits**
* **Tag:** --- Total Length of Address – Length of Offset – Length of Index

**Memory Access Logic**

* Row Access: 11 lines + RAS -> select 2048 rows
* Col Access: 11 lines + CAS -> select 2048 columns
* Read/write: controlled by WE / OE pins.

**Error Detection and Correction**

* Codeword = original data + parity bits
* D\_min = minimum hamming distance among codewords
* **Max Detectable Error: D\_min – 1 bits**
* **Max Correctable Error: Floor ( (D\_min – 1 ) / 2) bits**

**Contiguous Memory Allocation**

**Fixed Partition**

* Memory divided into one or more fixed-sized pattern

**Variable Parition**

* Division not fixed size (process allocated **just enough** memory), thus no internal fragmentation

**Fragmentation**

* Internal: unused space **within** partition (fixed partition only)
* External: enough total space, but non-contiguous
* A table with text on it

  Description automatically generatedSolution: compaction (defragmentation)

**Non-Contiguous Memory Allocation**

**Paging**

* **Page Table:** translate page -> Frames
* **Page:** logical memory divided into fixed-size blocks (process view)
* **Frame:** physical mem divided into fixed-size blocks (actual)
* **Virtual Address = { Page Number, Page Offset }**
* Page Offset has (page & frame size)

**Segmentation**

* Program divided into variable-szied segments
* **Virtual Address = { Segment Number, Segment Offset }**
* **Segment Table:**
* Base Address : Physical starting address of the segment
* Segment Limit: Length of that segment
* Physical Address = Base Address + Segment Offset

A diagram of a color wheel

Description automatically generated with medium confidence**Magnetic Disk**

* Constant Angular Velocity (CAV) – **LEFT**
* Multiple Zoned Recording -- **RIGHT**

A diagram of a circular object with text

Description automatically generated

* Access Must Specify: Cylinder, Head, Sector, Transfer Size, Memory Address
* **Access Delay = Seek Time + Rotational Delay**
* **Format of a Track** [ Sector ] [ Sector ] [Sector ] …
* **Format of Sector** [gap] [ID field] [gap] [Data Field] [gap]
* **ID Field:** [Synth Byte] [Track] [Head] [Sector] [CRC]

**Solid-State Drive (SSD)**

* Erase entire block before write
* Write data in new space then garbage collect ‘deleted’ space

**I/O Modules - Function**

* **Control and Timing**: Proc request; I/OM check & return device status; transfer data if device ready
* **Processor Communication: Command** (control bus, CPU -> I/OM), **Data** (data bus), **Status** (BUSY, READY, error conditions)
* **Device Communication: I/O -> devices.** Unique address for each device (depending on Memory Mapped I/O or Isolated I/O)
* **Data Buffering:** I/OM buffers data to handle different transfer rates.
* **Error Detection:** soft error (parity bit) OR hard error (paper jam)

**I/O Modules – Technique**

**Programmed I/O (PIO):** I/O Device -> CPU -> RAM

* CPU -> IOM **sends command**
* IOM perform action, sets **status register** when finished
* **Pooling:** CPU periodically checks status register to check completion

**Interrupt-Driven I/O:** I/O Device -> CPU -> RAM

* Same as PIO. Except after command send, CPU **continue other work**
* IOM perform action, **interrupt** CPU when ready
* CPU save state (register, PC, etc), process interrupt, restore state.

**Direct Memory Access (DMA):** I/O device -> DMA -> RAM

* CPU tells DMA what to do (e.g., device addr, memory loc, words)
* DMA transfer entire block of data. Interrupt CPU when finished.
* **Cycle Stealing:** DMA *steals* some cycles from CPU. CPU slowed.

A black and white diagram of a rectangular object

Description automatically generated

* **A diagram of a diagram

  Description automatically generatedSingle bus, detached DMA.** Use bus twice
* CPU suspend twice
* **Single bus, integrated DMA.** Use bus once
* **A diagram of a system

  Description automatically generated**CPU suspend once
* **Separate I/O Bus.**
* CPU suspended once
* Use Bus once