CS2100 Finals Cheatsheet v1.3 (2019-02-10)
by Julius Putra Tanu Setiaji, page 1 of 2

1 C Programming Language

1.1 Data Types

Type	sizeof	range
`int`	4 bytes	2s complement, (-2^{31}) to $(2^{31}-1)$ OR (-2,147,483,648 to 2,147,483,647)
`float`	4 bytes	1-bit sign, 8-bit exponent (excess-127), 23-bit mantissa
`double`	8 bytes	1-bit sign, 11-bit exponent (excess-1023), 52-bit mantissa
`char`	1 byte	ASCII (7 bits + 1 parity bit), A is 100 0001

Note that for mantissa there is an implicit leading bit 1

1.2 Format Specifiers

%f	`float`	`scanf`
%lf	`double`	`scanf`
%e	scientific notation	`printf`
%p	pointers	`printf`

1.3 Escape Sequences

\n	new line	\"	double-quote "
\t	tab	%%	percent %

1.4 Pointers

- New unary operators: `*` and &
- Convention: `int *abc;` AND `void f(int *);`

1.5 Functions

- Function prototype (just the type): e.g. `void g(int, int);`
- Variable scoping: by **functions**

1.6 Arrays

- When initialised with fewer values than elements, the rest are initialised as 0 (for `int`).
- Equivalence: value: `*(arr+2) == arr[2]` and memory location: `arr + 2 == &arr[2]`

1.7 String

- An array of characters, terminated by a null character `'\0'`
- Initialising: `char str[4] = "egg";` or `char str[4] = {'e', 'g', 'g', '\0'};`
- Read from stdin: `fgets(str, size, stdin); // reads until (size - 1) or '\n'` and `scanf("%s", str); // reads until whitespace` (note: `fgets` also reads in `'\n'`)
- Print to stdout: `puts(str);` ≡ `printf("%s\n", str);`
- String functions:
  - `strlen(s)`: returns the no of chars in s
  - `strcmp(s1, s2)`: compare ASCII val of correspndg chars, similar to Java `compareTo`
  - `strncmp(s1, s2, n)`: compare first n chars of s1 and s2
  - `strcpy(s1, s2)`: copy the string pointed by s2 into array pointed by s1, returns s1. E.g. `char s[4]; strcpy(s, "asdfgh"); // s == {'a', 's', 'd', '\0'};`
  - `strncpt(s1, s2, n)`: copy the first n chars of s2 to s1

1.8 Structures

- Structures allow grouping of heterogeneous members of different types.
- Assignment `result2 = result1;` copies the entire structure.
- Passing structure to function: the entire structure is copied.
- Alternatively, to change original structure, one can use pointer. Syntactic sugar: `(*player_ptr).name == player_ptr->name;`
- E.g.:

```
typedef struct {
^^Iint day, month, year;
} date_t;
typedef struct {
^^Iint stuNum;
^^Idate_t birthday;
} student_t;
student_t s1 = {1049858, {31, 12, 2020}};
// s1.birthday.month == 2020
```

1.9 Code Compilation Process

C Program (.c) -> **Preprocessor** -> Preprocessed code (.i) -> **Compiler** -> Assembly code (.asm) -> **Assembler** -> Object code (.o) -> **Linker** -> Executable (.hex)

2 Numbering Systems

2.1 Data Representation

- 1 byte = 8 bits
- n bits can represent up to 2^n values. Thus, to present m values, $\lceil \log_2 m \rceil$ is required

2.2 Decimal to Binary Conversion

- For whole numbers: repeated division-by-2 (look at remainder)
- For fractions: repeated multiplication-by-2 (look at "quotient")

2.3 Representation of Signed Binary Numbers

	Negation	Range	Zeroes
Sign-and-Magnitude	invert the sign bit (leading bit)	$-(2^{n-1}-1)$ to $2^{n-1}-1$	$+0_{10}$ and -0_{10}
1s Complement	invert all the bits	$-(2^{n-1}-1)$ to $2^{n-1}-1$	$+0_{10}$ and -0_{10}
2s Complement	invert all the bits, then add 1	-2^{n-1} to $2^{n-1}-1$	$+0_{10}$

For all of the above, the MSB (Most Significant Bit) represents sign.

2.3.1 Sign-and-Magnitude

- Range (8-bit): (1111 1111) to (0111 1111) = -127_{10} to $+127_{10}$
- Zeroes: 0000 0000 = $+0_{10}$ and 1000 0000 = -0_{10}
- e.g. $(0011\ 0100)_{sm} = +011\ 0100_2 = +52_{10}$, $(1001\ 0011)_{sm} = -(001\ 0011)_2 = -(19)_{10}$

2.3.2 1s Complement (Diminished Radix)

- Negation: $-x = 2^n - x - 1$
- Range (8-bit): (1000 0000) to (0111 1111) = -127_{10} to $+127_{10}$
- Zeroes: (0000 0000) = $+0_{10}$ and (1111 1111) = -0_{10}
- e.g. $(0000\ 1110)_{1s} = (0000\ 1110)_2 = (14)_{10}$, $(1111\ 0001)_{1s} = -(0000\ 1110)_2 = -(14)_{10}$

2.3.3 2s Complement (Radix complement)

- Negation: $-x = 2^n - x$
- Range (8-bit): (1000 0000) = -128_{10} to (0111 1111) = $+127_{10}$
- Zero: (0000 0000) = $+0_{10}$
- e.g. $(0000\ 1110)_{2s} = (0000\ 1110)_2 = (14)_{10}$, $(1111\ 0010)_{2s} = -(0000\ 1110)_2 = -(14)_{10}$

2.3.4 Excess-k

- Also known as offset binary. Use 0000 to represent $-k$ (lowest number possible)
- For unsigned, with n-bit number, $k = 2^{n-1} - 1$

2.3.5 Comparison

Value	Sign-&-Magnitude	1s Complement	2s Complement	Excess-8
+7	0111	0111	0111	1111
+6	0110	0110	0110	1110
+5	0101	0101	0101	1101
+4	0100	0100	0100	1100
+3	0011	0011	0011	1011
+2	0010	0010	0010	1010
+1	0001	0001	0001	1001
+0	0000	0000	0000	1000
-0	1000	1111	-	-
-1	1001	1110	1111	0111
-2	1010	1101	1110	0110
-3	1011	1100	1101	0101
-4	1100	1011	1100	0100
-5	1101	1010	1011	0011
-6	1110	1001	1010	0010
-7	1111	1000	1001	0001
-8	-	-	1000	0000

2.4 Operation on binary numbers

Algo for **Subtraction**: $A - B = A + (-B)$
Algo for **Overflow Check**: if MSB of first and second are the same, then MSB of resulting numbers must be the same too.

2.4.1 2s Complement on Addition

Algo: Perform binary addition; Ignore the carry out of the MSB; Check for overflow.

```
Example, 2s Complement 4-bit
+3  0011          -2    1110           -3     1101
+4  0100          -6    1010           -6     1010
--- -----         ---   -----          ---    -----
+7  0111 (No of)  -8 (1)1000 (No of)   -9  (1)0111 (Of!)
```

2.4.2 1s Complement on Addition

Algo: Perform binary addition; If there is carry out of the MSB, add 1 to the result; Check for overflow.

```
Example, 1s Complement 4-bit
+3  0011          -2    1101           -3     1100
+4  0100          -5    1010           -6     1001
--- -----         ---   -----          ---    -----
+7  0111 (No of)  -7 (1)0111           -9  (1)0101
                            1                    1
                        -----                -----
                         1000 (No of)     0110 (Of!)
```

2.5 Floating Point

- Single precision 32 bits: 1-bit sign, 8-bit exponent (excess-127), 23-bit mantissa
- Double precision 64 bits: 1-bit sign, 11-bit exponent (excess-1023), 52-bit mantissa
- e.g. $-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$, Exponent (excess-127) $= 2 + 127 = 129 = 1000\ 0001_2$

```
Sign Exponent          Mantissa
 1    10000001  10100000000000000000000
```

Hence, $1100\ 0000\ 1101\ 0000\ 0000\ 0000\ 0000\ 0000_2 = C0D00000_{16}$ (as `float`$= -6.5$, as `int`$= -1,060,110,336$)

3 MIPS

3.1 Loading a 32-bit constant into a register

1. Use `lui` to set the upper 16-bit: `lui $t0, 0xAAAA`
2. Use `ori` to set the lower-order bits: `ori $t0, $t0, 0xF0F0`

3.2 Memory Organisation

- Each address contains 1 byte = 8 bit of content.
- Memory addresses are 32-bit long (2^{30} memory words).
- 32 registers, each 4-byte long. Each word is also 4-byte long.

3.3 MIPS Instruction Classification

3.3.1 R-format

- `op $rd, $rs, $rt`
- `sll $rd, $rt, shamt` (rs = 0)

3.3.2 I-format

- `op $rt, $rs, Immediate`
- Displacement address: offset from address in rs
- PC-relative address: no of instructions from next instruction $PC = (PC + immediate) \times 4$

3.3.3 J-format

- `op Immediate`
- pseudo-direct address: remove last 2 bit (since word-aligned, by default the 2 least significant bits are 00) and 4 most significant bits (always the same as instruction address).
- eg xxxx00001111000011110000111100~~00~~, immediate is 00001111000011110000111100

4 Instruction Set Architecture

For modern processors: **General-Purpose Register** (GPR) is most common. **RISC** typically uses **Register-Register (Load/Store)** design, e.g. MIPS, ARM. **CISC** use a mixture of Register-Register and Register-Memory, e.g. IA32

4.1 Data Storage

- **Stack architecture**: Operands are implicitly on top of the stack.
- **Accumulator architecture**: One operand is implicitly in the accumulator (a special register)
- **General-purpose register architecture**: only explicit operands
  - **Register-memory architecture**: one operand in memory.
  - **Register-register (or load store) architecture**
- **Memory-memory architecture**: all operands in memory.

4.2 Memory Addressing Modes

- **Endianness**:
  - **Big-endian**: Most significant **byte** in lowest address
  - **Little-endian**: LS **byte** in lowest address ("reverse-order")
- **Addressing modes**: in MIPS, only 3: **Register** `add $t1, $t2, $t3`, **Immediate** `addi $t1, $t2, 98`, **Displacement** `lw $t1, 20($t2)`

4.3 Operations in the instruction set

Amdahl's law: make common cases fast. Optimise frequently used instructions (**Load**: 22%, **Conditional Branch**: 20%, **Compare** 16%, **Store**: 12%)

4.4 Instruction Formats

- **Instruction Length** :
  - **Variable-length instructions** : Require multi-step fetch and decode. Allow for a more flexible (but complex) and compact instruction set.
  - **Fixed-length instructions** : used in most RISC, e.g. MIPS instructions are 4-bytes long. Allow for easy fetch and decode, simplify pipelining and parallelism. Instruction bits are scarce.
  - **Hybrid instructions** : mix of var- and fixed-length instructions.
- **Instruction Fields** : **opcode** (unique code to specify the desired operation) and **operands** (zero or more additional information needed for the operation)

4.5 Encoding the Instruction Set

- **Expanding Opcode** scheme:
  - E.g. **Type-A**: **6-bit** opcode, **Type-B**: **11-bits** opcode. Max no of instructions $= 1 + (2^6 - 1) \times 2^5 = 2017$

    (1 Type-A instruction, Type-B "steals" $[2^6 - 1]$ opcodes from Type-A to prefix, each prefix having $[2^{11-6} = 2^5]$ opcodes)

5 Datapath

5.1 Instruction Execution Cycle in MIPS

- **Fetch** : Get instruction from memory, address is in Program Counter (PC) Register
- **Decode** : Find out the operation required
- **Operand Fetch** : Get operand(s) needed for operation
- **Execute** : Perform the required oepration
- **Result Write (Store)** : Store the result of the operation

5.2 Elements

- **Adder** **Input**: two 32-bit numbers, **Output**: sum of input
- **Register File** **Input**: three 5-bit: Read register 1, Read register 2, Write register; 32-bit Write data, **Output**: two 32-bit Read data 1, Read data 2; **Control**: 1-bit RegWrite (1 = write)
- **Multiplexer** **Input**: n lines of same width, **Control**: m bits where $n = 2^m$, **Output**: Select i^{th} input line if control $= i$
- **Arithmetic Logic Unit** : **Input**: two 32-bit numbers, **Control**: 4-bit to decide the particular operation, **Output**: 32-bit ALU result, 1-bit isZero?

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

- **Data Memory** **Input**: 32-bit memory address, 32-bit write data; **Control**: 1-bit MemWrite, 1-bit MemRead; **Output**: 32-bit Read-Data

6 Control

`Control Signal` **(Execution Stage)**

6.1 `RegDst` (Decode / Operand Fetch)

Purpose: Select the destination register number
- WR = **0**: `Inst[20:16]`, **1**: `Inst[15:11]`

6.2 `RegWrite` (Decode / Operand Fetch, RegWrite)

Purpose: Enable the writing of register
- **0**: No register write, **1**: New value will be written

6.3 `ALUSrc` (ALU) – Select the 2nd operand for ALU

- Operand2 = **0**: Register RD2, **1**: `SignExt(Inst[15:0])`

6.4 `ALUOp` (ALU) – help in generation of `ALUControl` signal.

- Instruction type **00** : `lw / sw`, **01** : beq, **10** : `R-type`

6.5 `ALUControl` (ALU) – Select the operation to be performed.

- Function = **0000** : AND, **0001** : OR, **0010** : add, **0110** : subtract, **0111** : slt, **1100** : NOR

6.6 `MemRead / MemWrite` (Memory) – enable r/w of data memory

- **0**: No mem r/w, **1**: r: from `Address` / w: `mem[Address] <- RD2`

6.7 `MemToReg` (RegWrite) – MUX input **swapped**

Purpose: Select the result to be written back to register file
- **1**: Register write data = Memory read data, **0**: ALU result

6.8 `PCSrc` (Memory / RegWrite) – PCSrc = (Branch AND isZero)

Purpose: Select the next PC value
- Next PC = **0**: PC + 4, **1**: `SignExt(Inst[15:0]) << 2` $+ (PC + 4)$

6.9 `Branch` (Memory / RegWrite) – whether a branch instruction.

	R-type	lw	sw	beq
RegDst	1	0	X	X
ALUSrc	0	1	1	0
MemToReg	0	1	X	X
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1
ALUop	10	00	00	01

7 Boolean Algebra

- AND $\cdot$, OR $+$, NOT $= '$, order: NOT -> AND -> OR

7.1 Laws

- **Identity**: $A + 0 = 0 + A = A$, $A \cdot 1 = 1 \cdot A = A$
- **Inverse/Complement**: $A + A' = 1$, $A \cdot A' = 0$
- **Commutative**: $A + B = B + A$, $A \cdot B = B \cdot A$
- **Associative**: $A + (B + C) = (A + B) + C$, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- **Distributive**: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, $A + (B \cdot C) = (A + B) \cdot (A + C)$

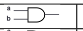
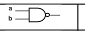
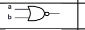
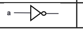
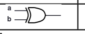
7.2 Duality

Swapping AND <-> OR, 1 <-> 0 keeps a statement valid.
- **idempotency**: $X + X = X$, $X \cdot X = X$
- **One/Zero element**: $X + 1 = 1$, $X \cdot 0 = 0$
- **Involution**: $(X')' = X$
- **Absorption**: $X + X \cdot Y = X$, $X \cdot (X + Y) = X$
- **Absorption (variant)**: $X + X' \cdot Y = X + Y$, $X \cdot (X' + Y) = X \cdot Y$
- **DeMorgan**: $(X + Y)' = X' \cdot Y'$, $(X \cdot Y)' = X' + Y'$
- **Concensus**: $X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$, $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$

CS2100 Finals Cheatsheet v1.3 (2019-02-10)
by Julius Putra Tanu Setiaji, page 2 of 2

7.3 Standard Forms
- Either **Sum-of-Product** (SOP) or **Product-of-sum** (POS)
- **minterm** of n vars: a product term with n literals from all the vars
- **maxterm** of n vars: a sum term with n literals from all the vars
- Generally with n vars: up to 2^n minterms and maxterms each.
- Each minterm is the **complement** of corrspndg maxterm
  e.g. $m2 = x \cdot y' = x' + y = M2$
- Conversion of standard forms, use DeMorgan
  e.g. $F = \sum m(1,4,5,6,7) = \prod M(0,2,3)$

8 Logic Circuits
8.1 Logic Gates

AND		$a \cdot b$	NAND		$(a \cdot b)'$
OR		$a + b$	NOR		$(a + b)'$
NOT		a'	XOR		$a \oplus b$

Note: $a \oplus b = (a \cdot b') + (a' \cdot b) = (a + b) \cdot (a \cdot b)'$

8.2 Universal Gates
- {NAND} is a complete set of logic.
  NOT: $(x \cdot x)' = x'$, **AND**: $((x \cdot y)' \cdot (x \cdot y)')' = ((x \cdot y)')' = x \cdot y$,
  OR: $((x \cdot x)' \cdot (y.y)')' = (x' \cdot y')' = (x')' + (y')' = x + y$
- {NOR} is a complete set of logic.
  NOT: $(x+x)' = x'$, **AND**: $((x+x)'+(y+y)')' = (x'+y')' = (x')' \cdot (y')' = x \cdot y$, **OR**: $((x+y)' + (x+y)')' = ((x+y)')' = x + y$

9 Simplification
9.1 Gray Code / Reflected Binary Code
- Only a single bit differs from one code value to the next.

0000 -> 0001 -> 0011 -> 0010 -> 0110 -> 0111 -> 0101 -> 0100 -> 1100 -> 1101 -> 1111 -> 1110 -> 1010 -> 1011 -> 1001 -> 1000

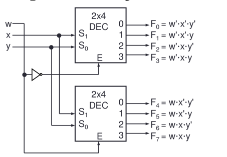
9.2 K-maps
- Labelled using gray code sequence.
- Recognise a group of size 2^n, including wrapovers.
- **Prime Implicant**: the biggest grouping possible (product term)
- **Essential Primt Implicant**: a PI that includes at least one minterm not covered by any other PI.
- Don't care – denoted with X

10 Combinatorial Circuits (Stateless)
10.1 Gate level design
- **Half-Adder** $C = X \cdot Y$, $S = X \oplus Y$
- **Full-adder** $C_{out} = X \cdot Y + (X \oplus Y) \cdot C_{in}$, $S = X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$
- 4-bit parallel adder by cascading 4 full-adders via their carries
- **6-person voting system** : Using 2 full-adders, each input for each vote. Result tallied using a parallel adder.
- **Magnitude Comparator** : **input**: 2 unsigned values A and B, **output**: "$A > B$", "$A = B$", "A<B"

10.2 Circuit Delays
- For each component, time = $\max(\forall t_{input}) + t_{\text{current component}}$
- Propagation delay of ripple-carry parallel adders $\propto$ no. of bits

11 MSI Components
11.1 Decoders
E.g. 2x4 decoder, $F_0 = X' \cdot Y'$, $F_1 = X' \cdot Y$, $F_2 = X \cdot Y'$, $F_3 = X \cdot Y$

X	Y	F_0	F_1	F_2	F_3
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**



- Larger decoders can be constructed from smaller ones.
- In general, for an n-bit code, a decoder could select up to 2^n lines.
- Also exists decoders with enable. $E = 0 \rightarrow F_0, F_1, F_2, F_3 = 0$
- Any combinatorial circuits with n inputs, m outputs can be implemented with an $n : 2^n$ decoder with m logic gates.
- Find sum-of-minterms, use each output of decoder corresponding to minterm/maxter as input to logic gate(s).
- Gate to use: Active-high: **OR** (minterm) **NOR** (maxterm), active-low: **NAND** (minterm) **AND** (maxterm)

11.2 Encoders
- 4-to-2 encoder, $D_0 = F_1 + F_3$, $D_1 = F_2 + F_3$
- 8-to-3 encoder, $x = D_4 + D_5 + D_6 + D_7$, $y = D_2 + D_3 + D_6 + D_7$, $z = D_1 + D_3 + D_5 + D_7$

8-to-3 encoder

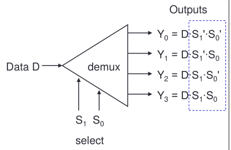
F_0	F_1	F_2	F_3	D_1	D_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1
otherwise				X	X

Priority encoder

F_0	F_1	F_2	F_3	D_1	D_0	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

11.3 Demultiplexers
1-to-4 demultiplexer

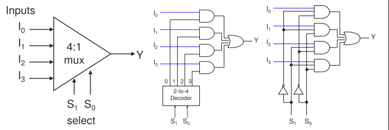
S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D



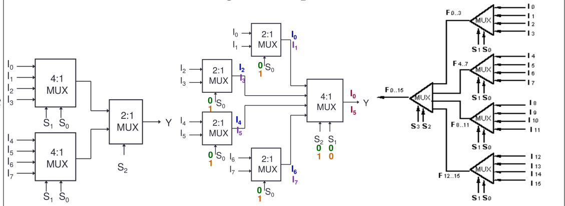
- Demux circuit is identical to a decoder w/ enable (data -> enable).

11.4 Multiplexers
4-to-1 multiplexer

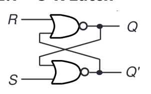
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



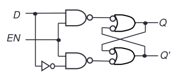
- $Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$
- In minterms, $Y = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$
- Possible to construct larger multiplexers from smaller ones.



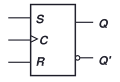
- Implement function using a 2^n-to-1 multiplexer: put '1' on the data for each minterm, '0' otherwise. Connect input to selector.
- Can also use a single smaller 2^{n-1}-to-1 multiplexer: reserve one of the input vars in data line, the rest for selection. Group input by selection line values in truth table.

12 Sequential Logic (Stateful)
12.1 S-R Latch


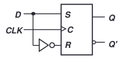
S	R	Q(t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	

12.2 D Latch


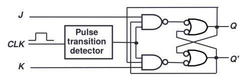
EN	D	Q(t+1)	
1	0	0	Reset
1	1	1	Set
0	X	Q(t)	No change

12.3 S-R Flip-flop


S	R	CLK	Q(t+1)	
0	0	X	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

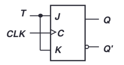
12.4 D Flip-Flop


D	CLK	Q(t+1)	
1	↑	1	Set
0	↑	0	Reset

12.5 J-K Flip-Flop


J	K	Q(t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

All CLK ↑

12.6 T Flip-Flop


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

12.7 Flip-Flop Excitation Tables
J-K Flip-Flop

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

S-R Flip-Flop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

D Flip-Flop

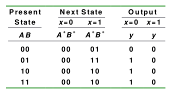
Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

T Flip-Flop

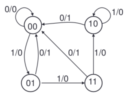
Q	Q^+	D
0	0	0
0	1	1
1	0	1
1	1	0

Async Inputs
PRE = HIGH $\rightarrow$ Q=HIGH
CLR = HIGH $\rightarrow$ Q=LOW

12.8 State Table & State Diagram
A, B (**Present State**)
x (**Input**)
A^+, B^+ (**Next State**)
y (**Output**)



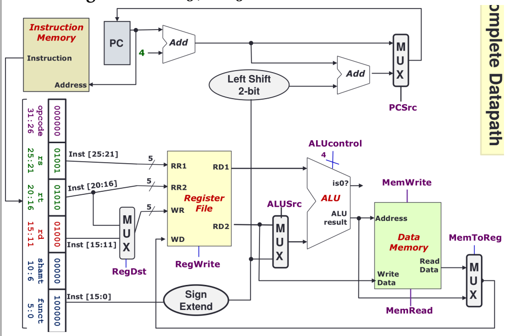
Present State	Next State		Output	
	x=0	x=1	x=0	x=1
AB	A⁺B⁺	A⁺B⁺	y	y
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0



13 Pipelining
5 execution stages: (1) **IF** (Instruction Fetch), (2) **ID** (Instruction Decode & Register Read), (3) **EX** (Execute / Calculate Address), (4) **MEM** (Access an operand in data memory), (5) **WB** (Write back result into a register)

13.1 Pipeline Datapath
Pipeline Registers store:
- **IF/ID** Instruction Read (including 2 Register Numbers for reading, 16-bit offset to be sign-extended to 32-bit); PC + 4
- **ID/EX** data values from register; 32-bit immediate value; PC + 4; Write Register Number
- **EX/ID** (PC + 4) + (Immediate ×4); ALU result; `isZero?` signal; Data Read 2 from register; Write Register Number
- **MEM/WB** ALU result; Memory read data; Write Register Number

Grouping:
- **EX stage**: `RegDst, ALUSrc, ALUop`
- **MEM stage**: `MemRead, MemWrite, Branch`
- **WB stage**: `MemToReg, RegWrite`



13.2 Performance calculations
T_k = Time for operation in stage k, N = no of stages
For I instructions, for total execution time ($time$) choose longest CT

13.3 Single Cycle Processor Performance
Cycle Time: $CT_{seq} = \sum_{k=1}^{N} T_k$, $Time_{seq} = I \times CT_{seq}$

13.4 Multi-Cycle Processor Performance (1 instruction many stages)
$CT_{multi} = max(T_k)$, $Time_{multi} = I \times Average\ CPI \times CT_{multi}$

13.5 Pipeline Processor
$CT_{pipeline} = max(T_k) + T_d$ (T_d = overhead for pipelining)
Cycles needed = $I + N - 1$ ($N - 1$ cycles wasted filling up the pipeline)
$Time_{pipeline} = (I + N - 1) \times (max(T_k) + T_d)$

Speedup = $\frac{Time_{seq}}{Time_{pipeline}}$

Ideal case assumption: every stage takes the same time ($\sum_{k=1}^{N} T_k = N \times T_k$), no pipeline overhead ($T_d = 0$), I much larger than N

13.6 Structural Hazard (simultaneous use of a hardware resource)
Soln: (1) **Stall** pipeline, (2) **Split** into **Data** & **Instruction** memory

13.7 Data Dependency: RAW (Read-after-Write)
Solution: **Forwarding & Bypass** (EX/MEM to ID/EX), but still problematic for LOAD instruction (need to stall)

13.8 Control Dependency
Decision only made in MEM stage (stall 3 cycles). Solution:
1. **Early Branch Resolution**, make decision in ID stage instead of MEM (but need dedicated adder), reduction 3 to 1 cycle delay, also add forward & bypass ALU to ID stage for data dependency, still problematic for LOAD instruction.
2. **Branch Prediction**, all branches assumed **not taken**. Correct guess no stall, wrong flush pipeline.
3. **Delayed branch**, move X non-control dependent instructions to after a branch as they are executed regardless of branch outcome. X (branch-delay slot) = 1 for MIPS

14 Cache
14.1 Memory Access Time
Hit rate, **Hit time** = cache access time, **miss rate** = 1 - hit rate, **miss penalty** = time to replace block cache + hit time
Average access time = hit rate × hit time + miss rate × miss penalty.

14.2 Write Policy
- **Write-through**, write data both to the cache & main memory. Problem: write will operate at speed of main memory. Solution: write buffer b/w cache and main memory.
- **Write-back** cache, only write to cache, write to main memory only when cache block is replaced (evicted). Problem: wasteful to write back every evicted cache. Solution: Use a "dirty bit" if cache content is changed. Write back only if dirty bit is set.

14.3 Types of Cache Misses
Compulsory misses on first access to a block, **Conflict misses** on collision, **Capacity misses** when blocks are discarded as cache is full

14.4 Handling Cache Misses
Read Miss: load data from memory to cache and then to register.
Write miss:
- **Write-allocate**: load complete block to cache, change the required word in cache, write to main memory (write policy).
- **Write-around**: no loading to cache, write to main memory only

14.5 Direct Mapped Cache – (index, valid, tag)
How to id: tag match with only 1 block.
Cache block size = 2^N bytes, **no of cache blocks** = 2^M
Offset = N bits, **Index** = M bits, **Tag** = $32 - (N + M)$ bits

14.6 N-way Set Associative Cache
How to id: tag match for all the blocks within the set.
A block maps to a unique set (each set having n "cache blocks")
Cache block size = 2^N bytes, **no of cache blocks** = $\frac{\text{size of cache}}{\text{size of block}}$,
no of sets = $\frac{\text{no of blocks}}{\text{n in n-way}} = 2^M$, **Offset** = N bits, **Set Index** = M bits

14.7 Fully Associative Cache – capacity miss, no conflict miss
How to id: tag match for all the blocks in the cache.
A memory block can be placed in any location in the cache.
Cache block size = 2^N bytes, **no of cache blocks** = 2^M
Offset = N bits, **Tag** = $32 - N$ bits (block number = tag)

14.8 Block Replacement Policy
- **Least Recently Used (LRU)**: Replace the block which has not been accessed for the longest time. For temporal locality. **Problem:** hard to keep track if there are many choices.
- **FIFO**, **Random Replacement**, **Least Frequently Used**

15 Performance
15.1 Execution Time
Avg Cycle/Instruction: **CPI** = $\frac{\text{(CPU time} \times \text{clock rate)}}{\text{Instruction count}} = \frac{\text{clock cycles}}{\text{instruction count}}$

CPU time = $seconds = instructions \times \frac{cycles}{instruction} \times \frac{seconds}{cycle}$

$CPI = \sum_{k=1}^{n} (CPI_k \times F_k)$ where $F_k = \frac{I_k \text{(Instruction frequency)}}{\text{Instruction count}}$

15.2 Amdahl's Law
Performance uis limited to the non-speedup portion of the program.
Execution time after improvement = Execution time of unaffected part + $\frac{\text{Execution time of affected part}}{\text{speedup}}$
Corollary: Optimise the common case first!

16 Additional stuffs
Opcode – maximise: allocate 1 each to the shortest opcode types.
Number of unusable bits: Type B=(32 - 29)=3, Type C=(32 - 16)=16
No of type A = $2^{32} - 2^3 - 2^{16} + 2$