Digital Design and Computer Architecture (DDCA)

Flavian Kaufmann
flkaufmann@ethz.ch
github.com/flavian112/ethz_ddca

Managing Complexity:

Abstraction:

Level	Example
Application Software	Programs
Operating Systems	Device Drivers
Architecture	Instructions, Registers
Microarchitecture	Datapath, Controllers
Logic	Adders, Memories
Digital Circuits	Gates
Analog Circuits	Amplifiers
Devices	Transistors, Diodes
Physics	Electrons

→ Hiding details when they are not important

Discipline:
→ Intentionally restricting design choices for greater productivity at higher abstraction level.

Three Y's:
- Hierarchy
  → A system is divided into modules of smaller complexity
- Modularity
  → Having well defined functions and interfaces
- Regularity
  → Encouraging uniformity, so modules can be easily reused

Circuit Design Goals:
Optimize...
- Area (proportional to cost)
- Speed/Throughput
- Power/Energy
- Design Time

Principles:
- Lazy, be creative
- Why?, take nothing for granted
- Engineering is not religion, use what works best for you.
- KISS (keep it simple and stupid), manage complexity.

Moore's Law:
Gordon Moore (1965): Number of transistors that can be manufactured doubles roughly every 18 months.

Binary Numbers

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
1	0	1	1	1	0	1	0	0	1	1	0	0	0	0	1

MSB … LSB

- Quad Word (16 bytes/128 bits)
- Double Word (8 bytes/64 bits)
- Word (4 bytes/32 bits)
- Half Word (2 bytes/16 bits)
- Byte (8 bits)
- Nibble (4 bits)
- Bit

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

n	2^n	IEC		SI
0	1			
1	2			
2	4			
3	8			
4	16			
5	32			
6	64			
7	128			
8	256			
9	512			
10	1024	Ki (kibi)	10^3	k (kilo)
11	2048			
12	4096			
13	8192			
14	16384			
15	32768			
16	65'536			
20		Mi (mebi)	10^6	M (mega)
30		Gi (gibi)	10^9	G (giga)
40		Ti (tebi)	10^{12}	T (tera)
50		Pi (pebi)	10^{15}	P (peta)

Range for N bit number: $0, 2^N-1$

Sign/Magnitude:
$-x = \{1^N | x\}$
Range for N bit number: $-2^{N-1}+1, 2^{N-1}-1$
Two zeroes, addition doesn't work.

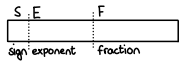
n	2^n
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625

One's complement:
$-x = \bar{x}$
Range for N bit number: $-2^{N-1}+1, 2^{N-1}-1$
Two zeroes, addition by end-around carry.

```
  0 0 0 1 0 0 0 1  (17)
+ 1 1 1 1 0 1 1 1  (-8)
  1 0 0 0 1 0 0 0
+               1
  0 0 0 1 0 0 0 1
```

Two's complement:
$-x = \bar{x}+1$, ignoring overflow
Range for N bit number: $-2^{N-1}, 2^{N-1}-1$
One zero, addition like unsigned addition

IEEE-754 Floating Point:

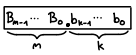


$(-1)^S \cdot 1.F \cdot 2^{E-bias}$ (1.F: Mantissa)

Name	Size (bits)	Mantissa (bits)	Exponent (bits)	Exponent Bias	Max	Min (pos)
Half precision	16	10+1	5	15	$6.55 \cdot 10^4$	$6.10 \cdot 10^{-5}$
Single precision	32	23+1	8	127	$3.40 \cdot 10^{38}$	$1.18 \cdot 10^{-38}$
Double precision	64	52+1	11	1023	$1.80 \cdot 10^{308}$	$2.23 \cdot 10^{-308}$

Special Values
- ±0: S=0/1, E=0, M=0
- ±Inf: S=0/1, E=1...1, M=0
- Nan: E=1...1, M≠0

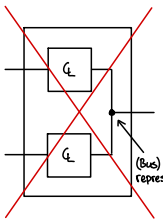
Fixed Point:
- m integer bits, k fraction bits

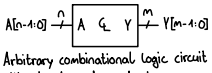


Addition:
1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponent
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble back into floating point format

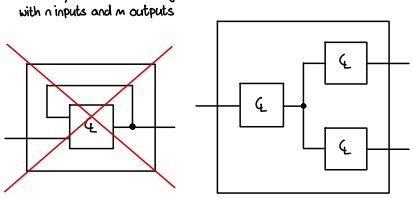
Combinational Circuits

A circuit is combinatorial if it consists of interconnected circuit elements such that:
- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once





Arbitrary combinational logic circuit with n inputs and m outputs

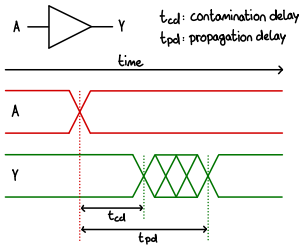


Each output Y_j for $j \in \{0,...,m-1\}$ is a function F_j of the inputs A_i for $i \in \{0,...,n-1\}$:

$$Y_j = F_j(A_0,..,A_{n-1})$$

Each output can also be represented as a truth table

Timing



t_{cd}: contamination delay
t_{pd}: propagation delay

Truth Table

A	Y
0	1
1	0

A	B	Y
0	0	0
0	1	1
1	0	X
1	1	0

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A truth table (n inputs) has 1 output and 2^n rows.

Don't care, can be 0 or 1.

Terminology

Name	Description	Examples
Literal	A variable or its complement	$A, \bar{B}$
Product/Implicant	And composition of literals	$A, \bar{B}, A \wedge A, \bar{A} \wedge B \wedge \bar{C}$
Minterm	Product that contains all of the inputs	$A \wedge B \wedge C, A \wedge \bar{B} \wedge \bar{C}$
Sum	Or composition of literals	$B \vee C, \bar{A}, A \vee \bar{B}$
Maxterm	Sum that contains all of the inputs	$\bar{A} \vee B \vee C, \bar{A} \vee \bar{B} \vee \bar{C}$

Sum-of-Products Canonical Form

The SOP canonical form is the sum of all minterms where the output Y is 1.

A	B	Y	Minterm	Minterm name
0	0	0	$\bar{A} \wedge \bar{B}$	m_0
0	1	1	$\bar{A} \wedge B$	m_1
1	0	0	$A \wedge \bar{B}$	m_2
1	1	1	$A \wedge B$	m_3

$Y = (\bar{A} \wedge B) \vee (A \wedge B)$
Sigma notation: $Y = F(A,B) = \sum(m_1, m_3)$

Product-of-Sums Canonical Form

The POS canonical form is the product of all maxterms where the output Y is 0.

A	B	Y	Maxterm	Maxterm name
0	0	0	$A \vee B$	M_0
0	1	1	$A \vee \bar{B}$	M_1
1	0	0	$\bar{A} \vee B$	M_2
1	1	1	$\bar{A} \vee \bar{B}$	M_3

$Y = (A \vee B) \wedge (\bar{A} \vee \bar{B})$
Pi notation: $Y = F(A,B) = \Pi(M_0, M_2)$

Boolean Algebra

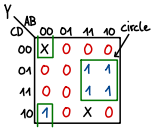
Axiom	Dual	Name
A1 $A \neq 1 \Rightarrow A = 0$	A1' $A \neq 0 \Rightarrow A = 1$	Binary Field
A2 $\bar{0} = 1$	A2' $\bar{1} = 0$	Not
A3 $0 \wedge 0 = 0$	A3' $1 \vee 1 = 1$	And/Or
A4 $1 \wedge 1 = 1$	A4' $0 \vee 0 = 0$	And/Or
A5 $0 \wedge 1 = 1 \wedge 0 = 0$	A5' $0 \vee 1 = 1 \vee 0 = 1$	And/Or

Theorem	Dual	Name
T1 $A \wedge 1 = A$	T1' $A \vee 0 = A$	Identity
T2 $A \wedge 0 = 0$	T2' $A \vee 1 = 1$	Null element
T3 $A \wedge A = A$	T3' $A \vee A = A$	Idempotency
T4 $\bar{\bar{A}} = A$		Involution
T5 $A \wedge \bar{A} = 0$	T5' $A \vee \bar{A} = 1$	Complements

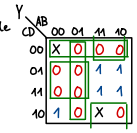
Theorem (multi variable)	Dual	Name
T6 $A \wedge B = B \wedge A$	T6' $A \vee B = B \vee A$	Commutativity
T7 $(A \wedge B) \wedge C = A \wedge (B \wedge C)$	T7' $(A \vee B) \vee C = A \vee (B \vee C)$	Associativity
T8 $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$	T8' $(A \vee B) \wedge (A \vee C) = A \vee (B \wedge C)$	Distributivity
T9 $A \wedge (A \vee B) = A$	T9' $A \vee (A \wedge B) = A$	Covering
T10 $(A \wedge B) \vee (A \wedge \bar{B}) = A$	T10' $(A \vee B) \wedge (A \vee \bar{B}) = A$	Combining
T11 $(A \wedge B) \vee (\bar{A} \wedge C) \vee (B \wedge C) = (A \wedge B) \vee (\bar{A} \wedge C)$	T11' $(A \vee B) \wedge (\bar{A} \vee C) \wedge (B \vee C) = (A \vee B) \wedge (\bar{A} \vee C)$	Consensus
T12 $\overline{(A \wedge B)} = \bar{A} \vee \bar{B}$	T12' $\overline{(A \vee B)} = \bar{A} \wedge \bar{B}$	de Morgan

Karnaugh Maps

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1



$Y = (\bar{A} \wedge \bar{B} \wedge \bar{D}) \vee (A \wedge D)$

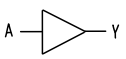


$Y = (A \vee \bar{D}) \wedge (C \vee D) \wedge (\bar{A} \vee D) \wedge (A \vee \bar{B})$

1. Use the fewest circles necessary to cover all the 1/0's.
2. All squares in each circle must only contain 1/0's or X's.
3. Each circle must span a rectangular block with side lengths that are a power of two.
4. Each circle should be as large as possible.
5. A circle may wrap around the edges of a K-map.
6. A 1/0 in a K-map may be circled multiple times if doing so allows fewer circles to be used.
7. Circles of 1s are products and circles of 0s are sums.

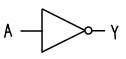
Logic Gates

Buffer:



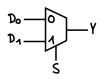
A	Y
0	0
1	1

Inverter (NOT):

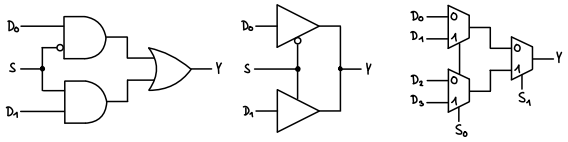


A	Y
0	1
1	0

Multiplexer (MUX):

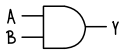


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



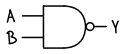
In general, wider (2^n:1) multiplexers can be achieved by two level logic, an array of tri-state buffers with a (n:2^n) decoder, or in a hierarchical design.

AND:



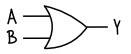
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

NAND:



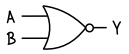
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

OR:



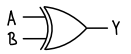
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOR:



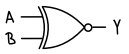
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XOR:



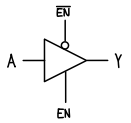
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

XNOR:



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

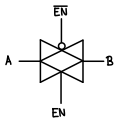
Tri-State Buffer:



EN	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

High Impedance

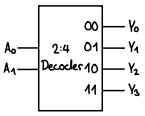
Transmission Gate:



EN	A	B
0	Z	Z
1	B	A

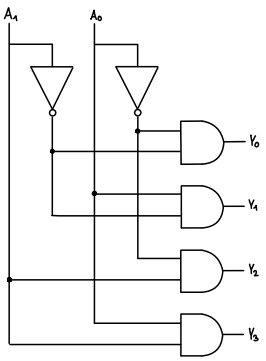
Combinational Building Blocks

Decoder (DEC):



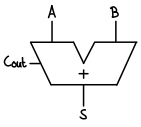
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

The outputs of a decoder are also the corresponding minterms.



Arithmetic

Half Adder:

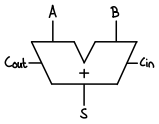


A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$Cout = A \wedge B$

Full Adder:



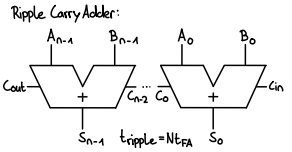
Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus Cin$

$Cout = (A \wedge B) \vee (A \wedge Cin) \vee (B \wedge Cin)$

Carry Propagation Adders (CPA):

Ripple Carry Adder:



$t_{ripple} = N t_{FA}$

Carry Lookahead Adder:

$P_i = A_i \vee B_i$

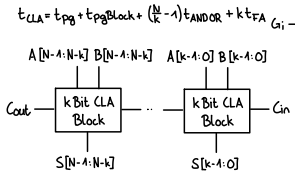
$G_i = A_i \wedge B_i$

$G_{3:0} = G_3 \vee (P_3 \wedge (G_2 \vee (P_2 \wedge (G_1 \vee (P_1 \wedge G_0)))))$

$P_{3:0} = P_3 \wedge P_2 \wedge P_1 \wedge P_0$

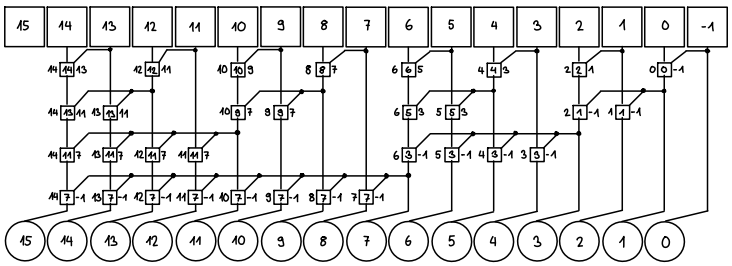
$C_i = G_{i:j} \vee (P_{i:j} \wedge C_j)$

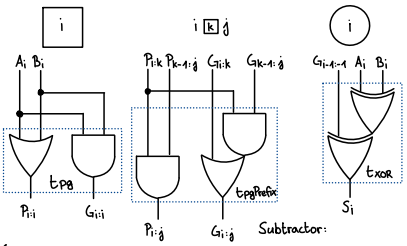
$t_{CLA} = t_{pg} + t_{pgBlock} + \left(\frac{N}{k} - 1\right) t_{ANDOR} + k\, t_{FA}$



For N>16 a CLA is generally faster, but still scales linearly with respect to propagation delay.

(Parallel) Prefix Adder (PPA):



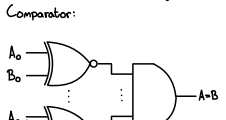


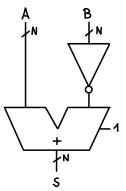
$G_{-1} = Cin$

$P_{-1} = 0$

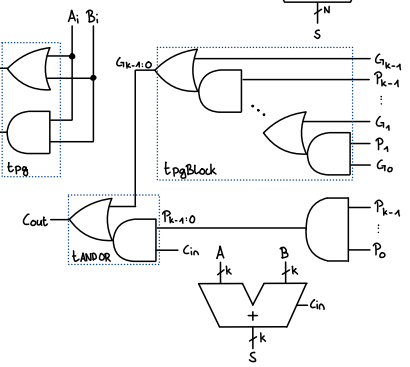
$t_{PA} = t_{pg} + \log_2(N)\, t_{pgPrefix} + t_{XOR}$

Comparator:

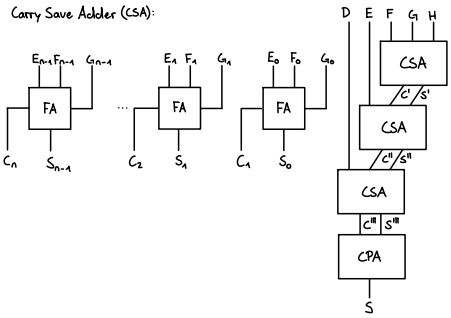




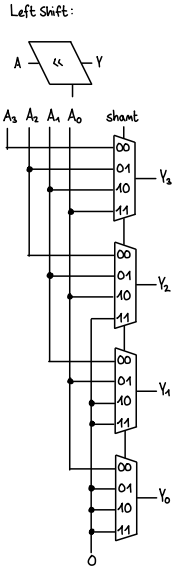
Subtractor:



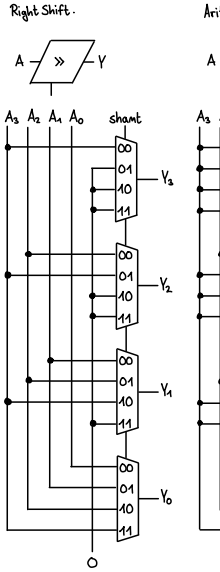
Carry Save Adder (CSA):



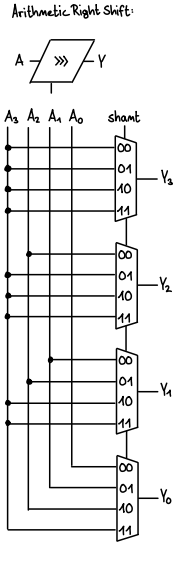
Left Shift:



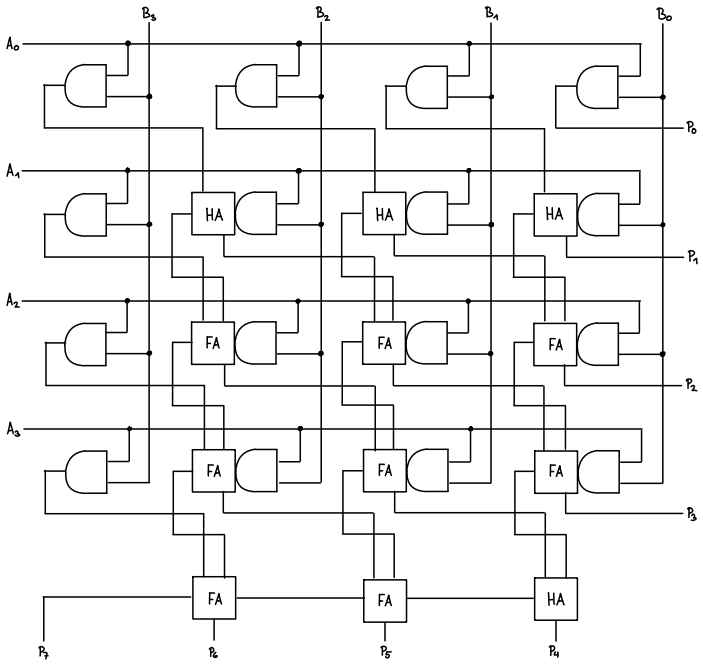
Right Shift:



Arithmetic Right Shift:



Parallel Multiplier:




```
        1 0 1 0
      x 0 1 0 1
      ---------
        1 0 1 0
      0 0 0 0
    1 0 1 0       Partial
+ 0 0 0 0         Products
  ---------------
  0 1 1 0 0 1 0 0
```

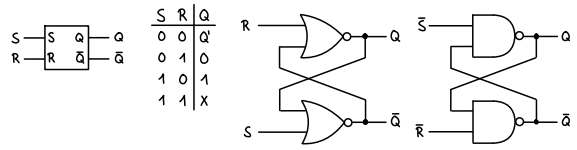
Synchronous Sequential Logic

Synchronous sequential logic circuits consist of interconnected circuit elements such that:

- 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.

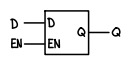
Sequential Building Blocks

SR Latch:

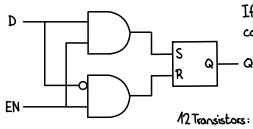


S	R	Q
0	0	Q'
0	1	0
1	0	1
1	1	X

D Latch:

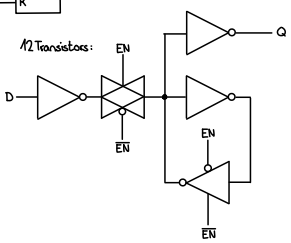


EN	D	Q
0	X	Q'
1	0	0
1	1	1

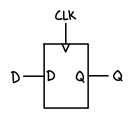


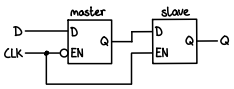
If enabled, the latch is called transparent, otherwise it is opaque.

12 Transistors:

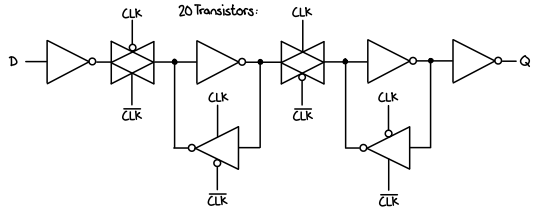


D Flip-Flop:





20 Transistors:



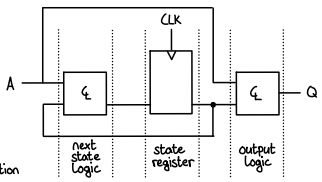
A register is an array of D Flip-Flops that all have the same clock signal.

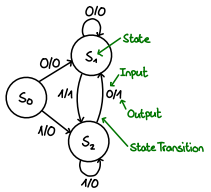
Finite State Machines

Mealy Machine:

$(S, S_0, \Sigma, \Lambda, T, G)$

- S: Finite set of states
- $S_0 \in S$: Initial State
- Σ: Input Alphabet
- Λ: Output Alphabet
- $T: S \times \Sigma \to S$: Transition Function
- $G: S \times \Sigma \to \Lambda$: Output Function

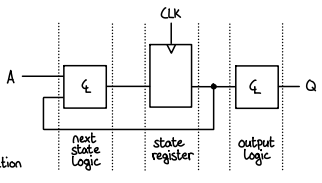


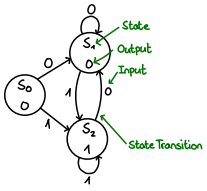


Moore Machine:

$(S, S_0, \Sigma, \Lambda, T, G)$

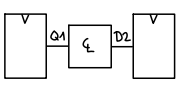
- S: Finite set of states
- $S_0 \in S$: Initial State
- Σ: Input Alphabet
- Λ: Output Alphabet
- $T: S \times \Sigma \to S$: Transition Function
- $G: S \to \Lambda$: Output Function



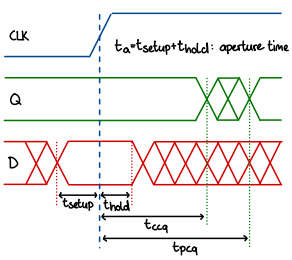


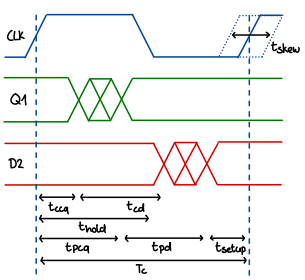
Timing

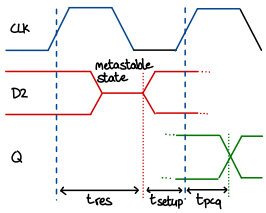
t_{setup}: setup time
t_{hold}: hold time
t_{ccq}: contamination delay
t_{pcq}: propagation delay

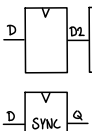


Setup time constraint:
$T_c \geq t_{pcq} + t_{pd} + t_{setup} (+ t_{skew})$
Hold time constraint:
$t_{cd} \geq t_{hold} - t_{ccq} (+ t_{skew})$









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

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

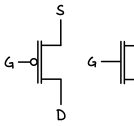
$MTBF = \frac{1}{P(\text{failure})/s}$

$P(\text{failure})/s = P(\text{failure}) \cdot N$, N: # of changes of D

Transistors

MOSFETs (Metal Oxide Semiconductor Field Effect Transistor):

PMOS: NMOS:



$P_{total} = P_{dynamic} + P_{static}$

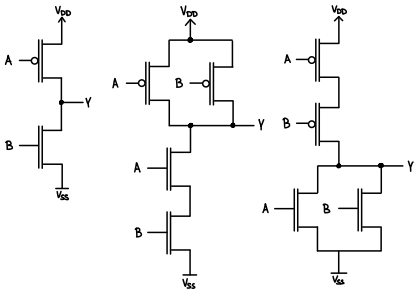
$P_{dynamic} = \frac{1}{2} C V_{DD}^2 f$

$P_{static} = I_{DD} V_{DD}$

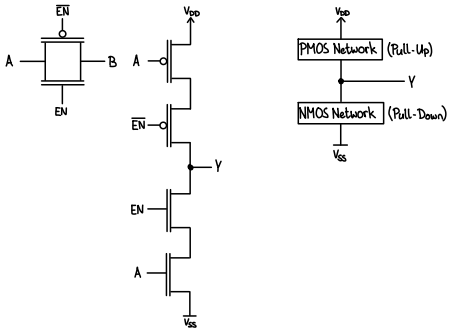
Saturation: $V_{GS} > V_{tn}$ and $V_{DS} \geq V_{GS} - V_{tn}$

$I_D \sim \frac{W}{L}$

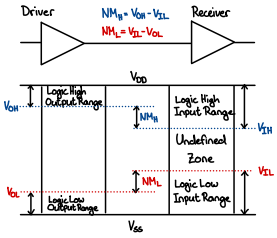
NOT: NAND: NOR:



Transmission Gate: Tri-State Buffer (inverted): General:

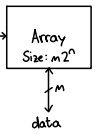


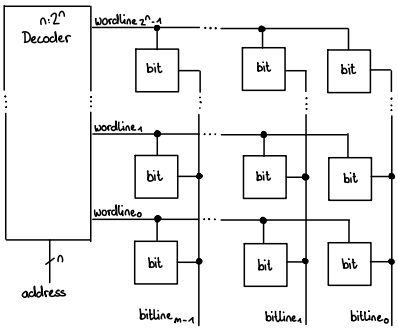
Logic Level, Noise Margins



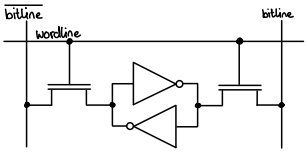
Memories

- Flip-Flops / Latches (Registers)
  - Very fast, parallel access
  - Expensive (20+ transistors per bit)
- SRAM (Static RAM)
  - relatively fast, only one data word at a time
  - Less expensive (6 transistors per bit)
- DRAM (Dynamic RAM)
  - slower, reading destroys content (refresh), one data word at a time
  - cheaper (1 transistor per bit)
- Other (HDD, Flash, ...)
  - much slower
  - no transistors involved, non-volatile

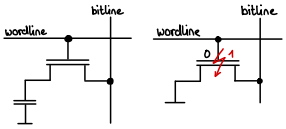




SRAM:



DRAM: ROM:



MIPS

Instruction Formats:

R-Type:

opcode	rs	rt	rd	shamt	funct
31 ···(6bits)··· 26	25 ···(5bits)··· 21	20 ···(5bits)··· 16	15 ···(5bits)··· 11	10 ···(5bits)··· 6	5 ···(6bits)·· 0

I-Type:

opcode	rs	rt	imm
31 ···(6bits)··· 26	25 ···(5bits)··· 21	20 ···(5bits)··· 16	15 ···(16 bits)··· 0

J-Type:

opcode	addr
31 ···(6bits)··· 26	25 ···(26bits)··· 0

Instructions:

Mnemonic	Name	Type	Operation	Ovf	Uns	Op	Fct
add	Add	R	rd = rs + rt	✓		0	20
addu	Add Unsgn	R	rd = rs + rt			0	21
addi	Add Imm	I	rt = rs + SignExtImm	✓		8	-
addiu	Add Imm Unsgn	I	rt = rs + SignExtImm			9	-
sub	Subtract	R	rd = rs - rt	✓		0	22
subu	Subtract Unsgn	R	rd = rs - rt			0	23
and	And	R	rd = rs & rt			0	24
andi	And Imm	I	rt = rs & ZeroExtImm			C	-
or	Or	R	rd = rs | rt			0	25
ori	Or Imm	I	rt = rs | ZeroExtImm			D	-
nor	Nor	R	rd = ~(rs|rt)			0	27
sll	Shift Left Logic	R	rd = rt << shamt			0	0
srl	Shift Right Logic	R	rd = rt >> shamt			0	2
sra	Shift Right Arithmetic	R	rd = rt >>> shamt			0	3
slt	Set Less Than	R	rd = rs < rt ? 1 : 0			0	2A
slti	Set Less Than Imm	I	rt = rs < SignExtImm ? 1 : 0			A	-
sltu	Set Less Than Unsgn	R	rd = rs < rt ? 1 : 0		✓	0	2B
sltiu	Set Less Than Imm Unsgn	I	rt = rs < SignExtImm ? 1 : 0		✓	B	-
beq	Branch on eq	I	if rs == rt : PC += 4 + BranchAddr			4	-
bne	Branch on not eq	I	if rs != rt : PC += 4 + BranchAddr			5	-
j	Jump	J	PC = JumpAddr			2	-
jal	Jump and Link	J	ra = 8+PC, PC = JumpAddr			3	-
jr	Jump Register	R	PC = rs			0	8
lw	Load Word	I	rt = M[rs + SignExtImm]			23	-
lhu	Load Half Unsgn	I	rt = {16'b0, (M[rs + SignExtImm])[15:0]}			25	-
lbu	Load Byte Unsgn	I	rt = {24'b0, (M[rs + SignExtImm])[7:0]}			24	-
lui	Load Upper Imm	I	rt = {imm, 16'b0}			F	-
sw	Store Word	I	M[rs + SignExtImm] = rt			2B	-
sh	Store Half	I	M[rs + SignExtImm][15:0] = rt[15:0]			29	-
sb	Store Byte	I	M[rs + SignExtImm][7:0] = rt[7:0]			28	-
mfhi	Move from Hi	R	rd = Hi			0	10
mflo	Move from Lo	R	rd = Lo			0	12
mult	Multiply	R	{Hi, Lo} = rs * rt			0	18
multu	Multiply Unsgn	R	{Hi, Lo} = rs * rt		✓	0	19

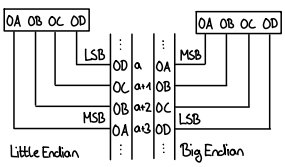
May generate overflow exception.
Operands are considered unsigned.

SignExtImm = {16{imm[15]}, imm}
ZeroExtImm = {16{1'b0}, imm}
BranchAddr = {14{imm[15]}, imm, 2'b0}
JumpAddr = {(4+PC)[31:28], addr, 2'b0}

RISC (Reduced Instruction Set Computer)
- small number of simple instructions
- eg. MIPS, RISC-V, ARM

CISC (Complex Instruction Set Computer)
- Large number of instructions
- eg. x86

Endianness:



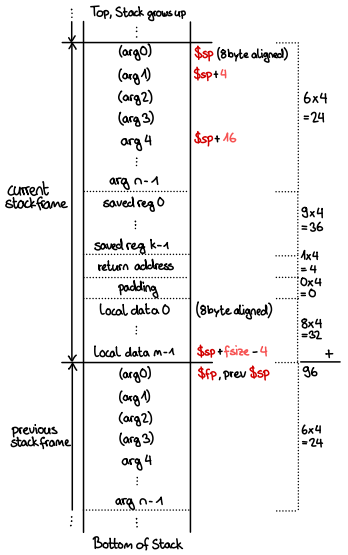
Registers:

Name	Idx	Use	callee saved
zero	0	Constant zero	-
at	1	Assembler temporary	x
v0-v1	2-3	Return values	x
a0-a3	4-7	Arguments	x
t0-t7	8-15	Temporaries	x
s0-s7	16-23	Saved	✓
t8-t9	24-25	Temporaries	x
k0-k1	26-27	Kernel	x
gp	28	Global Pointer	✓
sp	29	Stack Pointer	✓
fp	30	Frame Pointer	✓
ra	31	Return Address	✓

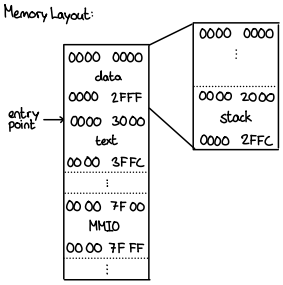
Assembly

```
.data
    staticVar: .word 0          # 4 bytes
.text
.global main
main:
    addi  $a0  $t0   0   #1st arg
    addi  $a1  $t1   0   #2nd arg
    addi  $a2  $t2   0   #3rd arg
    addi  $a3  $t3   0   #4th arg
    addi  $sp  $sp  -24  #increase stack for args
    sw    $t4  16($sp)   #5th arg
    sw    $t5  20($sp)   #6th arg
    jal   func           #call function
    addi  $sp  $sp  24   #decrease stack
    addi  $t6  $v0   0   #return value
end:
    j     end            #halt loop
func:
    addi  $sp  $sp  -96  #push stack frame   prologue
    sw    $ra  60($sp)   #save registers
    sw    $s0  24($sp)
          :
    sw    $s7  52($sp)
    sw    $fp  56($sp)
    addi  $fp  $sp  96   #set frame ptr
    addi  $t0  $a0   0   #1st arg            body
    addi  $t1  $a1   0   #2nd arg
    addi  $t2  $a2   0   #3rd arg
    addi  $t3  $a3   0   #4th arg
    lw    $t4  16($fp)   #5th arg
    lw    $t5  20($fp)   #6th arg
    # body
    addi  $v0  $t6   0   #return value       epilogue
    lw    $fp  56($sp)   #restore registers
    lw    $s7  52($sp)
          :
    lw    $s0  24($sp)
    lw    $ra  60($sp)
    addi  $sp  $sp  96   #pop stack frame
    jr    $ra            #return to caller
```

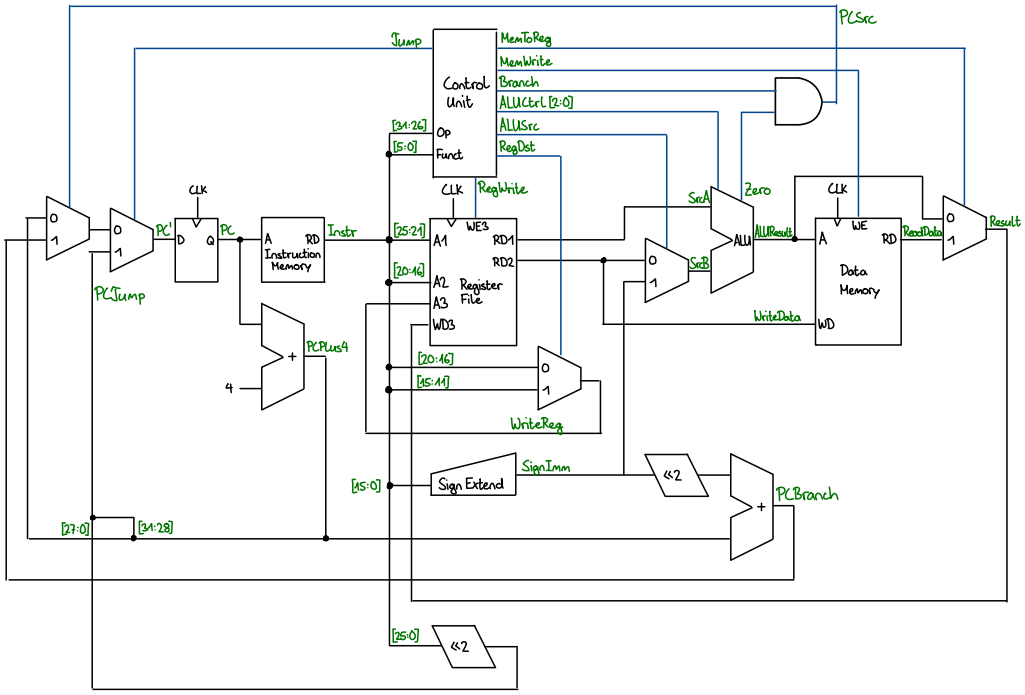
Stack frame:



Memory Layout:

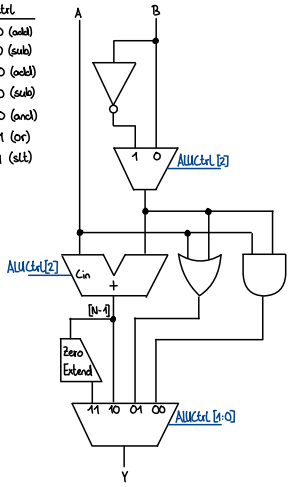


Single Cycle Microarchitecture



Instruction	Op	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUOp	Jump
R-Type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

ALUOp	Funct	ALUCtrl
00	X	010 (add)
01	X	110 (sub)
10	100000	010 (add)
10	100010	110 (sub)
10	100100	000 (and)
10	100101	001 (or)
10	101010	111 (slt)

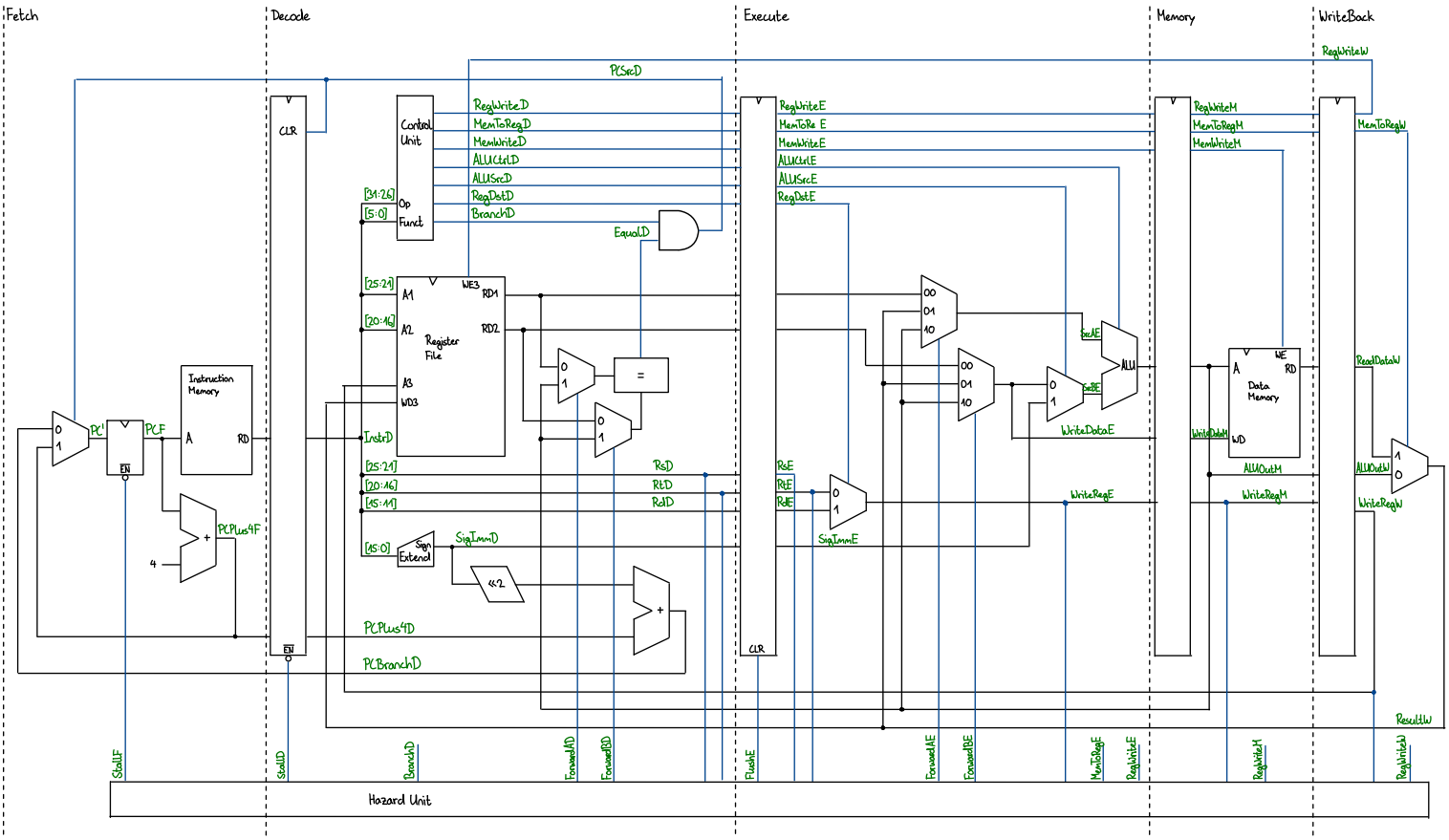


Critical Path:

$T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + 2t_{mux} + t_{RFsetup}$

Time to execute N Instructions = $N \cdot CPI \cdot \frac{1}{f_c} = N \cdot 1 \cdot \frac{1}{f_c}$

Pipelined Architecture



$T_c \geq \max($

$t_{pcq} + t_{mem} + t_{setup}$,	fetch
$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$,	decode
$t_{pcq} + t_{mux} + t_{ALU} + t_{setup}$,	execute
$t_{pcq} + t_{memwrite} + t_{setup}$,	memory
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	writeback

$)$

//Forwarding Logic:

```
assign ForwardAD = (RsD!=0) & (RsD == WriteRegM) & RegWriteM
assign ForwardBD = (RtD!=0) & (RtD == WriteRegM) & RegWriteM
```

//Stalling Logic:

```
assign lwstall = ((RsD == RtE) | (RtD == RtE)) & MemToRegE
assign branchstall = (BranchD & RegWriteE & (WriteRegE == RsD | WriteRegE == RtD))
                   | (BranchD & MemToRegM & (WriteRegM == RsD | WriteRegM == RtD))
```

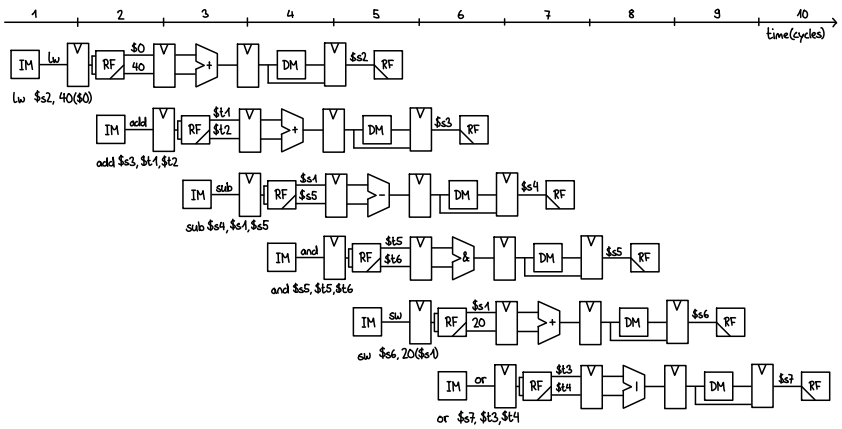
//Stall signals:

```
assign StallF, StallD, FlushE = lwstall | branchstall
```

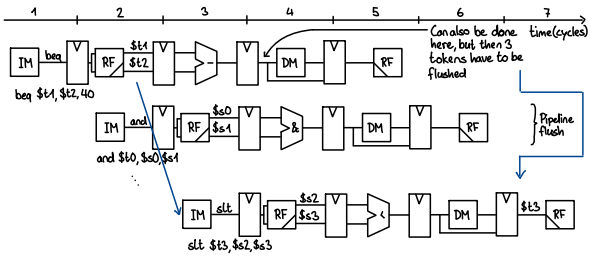
Hazards

- Data Hazard: Data for next instruction is not ready
- Control Hazard: Next Instruction is unknown

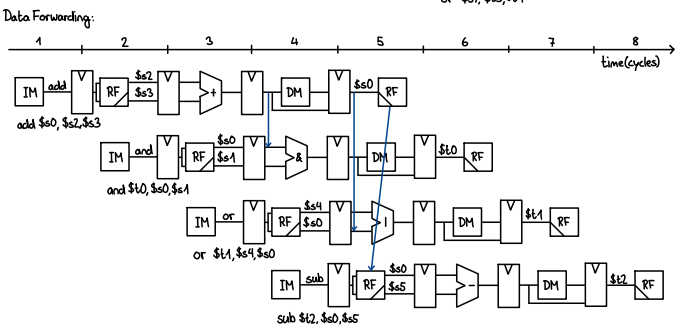
Pipeline:



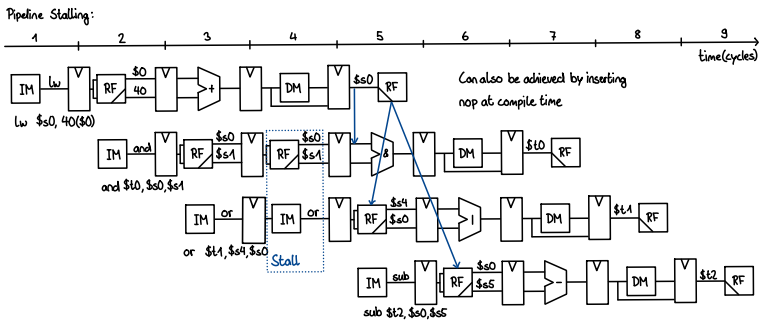
Branch Resolution:



Data Forwarding:



Pipeline Stalling:



Verilog:

Comments:

```
// One-liner
/* Multiple
   lines */
```

Numeric Constants:

```
8'b_0110_1010   // binary
8'o152          // octal
8'd106          // decimal
8'h6A           // hexadecimal
"j"             // ASCII

78'bz           // High Impedance
```

Nets and Variables:

```
wire [3:0]  w;      // assign outside always
reg  [1:7]  r;      // assign inside always
reg [7:0] mem [31:0];

integer j;          // compile time variable
genvar k;           // generate variable
```

Parameters:

```
parameter N=8;
localparam S= 2'd3;
```

Assignments:

```
assign Output = A*B;
assign {C,D} = {D[5:2],C[1:0],E};
```

Operators:

```
A[N], A[N:M]                    // Select
&A, ~&A, |A, ~|A, ^A, ~^A       // Reduction
!A, ~A                          // Complement
+A, -A                          // Unary
{A, ..., B}                     // Concatenate
{N{A}}                          // Replicate
A*B, A/B, A%B, A+B, A-B         // Arithmetic
A<<B, A>>B, A<<<B               // Shift
A>B, A<B, A>=B, A<=B, A==B, A!=B // Relation
A&B, A|B, A^B, A~^B             // Bitwise
A&&B, A||B                      // Logical
A?B:C                           // Conditional
```

Module:

```
module MyModule
#(parameter N=8)  //optional parameter
(input rst, clk,
 output [N-1:0] out);
    //impl
endmodule

MyModule #(.N(16)) mod (.rst(RST), .clk(CLK), .out(OUT));
```

Case:

```
always @(*) begin
    case (mux)
        2'b00 : A = 8'd13; //(blocking)
        2'b01,
        2'b10 : A = 8'd108;
        2'b11 : A = 8'd12;
    endcase
end

always@(*) begin
    casex (dec)
        4'b1xxx : enc = 2'b00;
        4'b01xx : enc = 2'b01;
        4'b001x : enc = 2'b10;
        4'b0001 : enc = 2'b11;
        default : enc = 2'b00;
    endcase
end
```

Generate:

```
genvar j;
wire [11:0]out [15:0]
generate
    for(j=0; j<20; j=j+1)
        begin: Gen_Modules
            MyModule #(.N(13)) mod (
                .rst(RST), .clk(CLK),
                .out(out[j])
            );
        end
endgenerate
```

Synchronous:

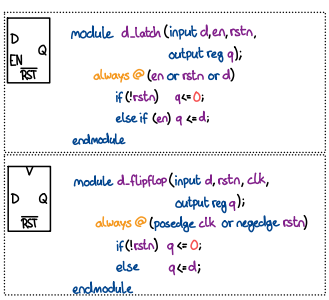
```
always @(posedge clk) begin
    if(rst)  B <= 0;        //(non-blocking)
    else     B <= B+ 1'b1;
end
```

Loop:

```
always @(*) begin
    count = 0;
    for(j=0; j<8; j=j+1)
        count = count+input[j];
end
```

Function:

```
function [5:0]F;
    input [3:0]A;
    input [2:0]B;
    begin
        F = {A+1'b1, B+2'd2};
    end
endfunction
```



State Machine:

```
reg [1:0]state
parameter  s0 = 2'b00,
           s1 = 2'b01,
           s2 = 2'b10,
           s3 = 2'b11;
always @(posedge clk) begin
    if (rst) state = s0;
    else case (state)
        s0: state = s1;
        s1: state = s2;
        s2: state = s3;
        s3: state = s0;
    endcase
end
```

Testbenches:

```
`timescale 1ns/1ps      //<unit time>/<resolution>
initial
always
#10                     // delay by 10ns
$display("%b bin, %h hex, %d dec, %o oct, %c ASCII, %s str, %t time",
          ..., $time)
$readmemh("filename", testvec)
$finish
```

Functional Verification