$2^{10} = 1024$

Digitaltechnik ZF

Knobelf

Invertiere & +1 ! $(\ldots)_2$!

Hexadecimal	unsigned Decimal	Binär	Two's complement	sign/magnitude
0	0	0000	0	0
1	1	0001	1	1
2	2	0010	2	2
3	3	0011	3	3
4	4	0100	4	4
5	5	0101	5	5
6	6	0110	6	6
7	7	0111	7	7
8	8	1000	-8	-0
9	9	1001	-7	-1
A	10	1010	-6	-2
B	11	1011	-5	-3
C	12	1100	-4	-4
D	13	1101	-3	-5
E	14	1110	-2	-6
F	15	1111	-1	-7

Two's complement ⟷ sign/magnitude

- \+ Easy addition/subtraction (just as normal)
- \+ Every combination unique

- \+0 / -0
- subtraction more difficult
- other circuits for negative numbers

How many bits to represent $(1000)_{10}$?

$2^{10} = 1024 \Rightarrow$ 10 bits [9:0] $\Rightarrow$ up to 1023

Hexadecimal: How many cycles?

Find difference of start and end address

$(1000)_{16} = 16^3 = 4^6 = 4 \cdot 2^{10} \Rightarrow 2^{10}$ cycles

tera-	T	10^{12}	10^{-12}	p	pico-
giga-	G	10^9	10^{-9}	n	nano-
mega-	M	10^6	10^{-6}	µ	micro-
kilo-	k	10^3	10^{-3}	m	milli-
hecto-	h	10^2	10^{-2}	c	centi-
deka-	da	10^1	10^{-1}	d	deci-

8 bit = 1 byte

$\boxed{\bar{A} \cdot B + D \Leftrightarrow \neg A \wedge B \vee D}$

SOP sum of products (Minterm) / **PoS** product of sums (Maxterm)

A	B	Y	
0	0	0	→ $A+B$
0	1	1	→ $\bar{A} \cdot B$
1	0	0	→ $\bar{A} + B$
1	1	1	→ $A \cdot B$

SoP: $Y = (\bar{A} \cdot B) + (A \cdot B)$

PoS: $Y = (A + B) \cdot (\bar{A} + B)$

Karnaugh Maps

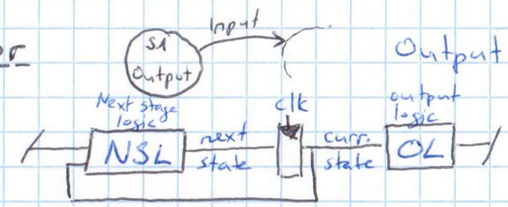
AB \ CD	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

SoP: $Y = (B \cdot D) + (\bar{B} \cdot \bar{D})$ ① $A \to A$

PoS: $Y = (B + \bar{D}) \cdot (\bar{B} + D)$ ⓪ $A \to \bar{A}$

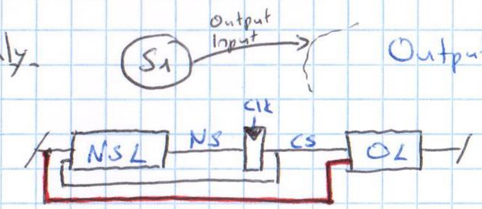
FSM Finite state Machine

Moor



Output depends only on current state.

Mealy



Output depends on current state and on Input

Diagram

1. Reset State
2. Names of States
3. Do not mix Moor & Mealy (Output!)
4. Output everywhere
5. Label transitions IN=00...
6. Define all cases for transitions

for 2 inputs a,b

a=1 & b=0

a=1 | b=0

+1 Bit

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192

Assembly Code

```
add/sub   $s0, $s1, $s2        # $s0 = $s1 ± $s2
lw/sw     $s0, 0($s1)          # load/store word
addi/subi $s0, $s1, 2          # add/sub integer
beq/bne   $s0, $s1, target     # branche if equal/not equal
sll/srl   $s0, $s1, 2          # shift left/right logical (*2^n)
sla/sra   $s0, $s1, 2          # shift left/right arithetically (negative)
j         target               # jump to target
jr        $ra                  # jump back to former routine (address in register)
lui       $s0, 0x1234          # load upper [31:16]
ori       $s0, 0x8000          # load lower [15:0]
target:
slt       $s0, $s1, $s2        # set less than [($s1 < $s2)?1:0;]
jal       name                 # jump and link (subroutine call (stack!))
```

$sp stack pointer: given by last routine

1. move forward (-4)
2. store result
3. move back (+4)

$ra address of previous routine

Speed: $T = N \cdot CPI \cdot \frac{1}{f}$

- T: time it takes
- N: instructions
- CPI: Cycles per instruction
- f: frequency

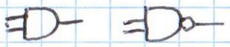
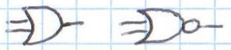
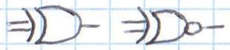
To return to former programm after many jumps (subroutines):

```
addi  $sp, $sp, -4     # shift sp
sw    $ra, 0($sp)
⋮
lw    $ra, 0($sp)
addi  $sp, $sp, 4      # shift sp back
jr    $ra
```

Single-Cycle: Perform each instruction in one-cycle
→ constant CPI (=1) no pipelining.

Multi-Cycle: Perform each instruction in multiple cycles
→ pipelining can increase throughput

Verilog

~	Not	
*, /, %	mult., div., mod	
+, −	add, sub	
<<, >>	shift	add zeros on both sides
<<<, >>>	arithmetic shift	zeros right side, old MSB right side
<, <=, >, >=, ==, !=	comparison	
&, ~&	AND, NAND	
|, ~|	OR, NOR	
^, ~^	XOR, XNOR	
	Buffer	
2'b10;	Binary number	
2'hF0;	Hex number	
<=	Non-blocking	end of always block
=	Blocking	imm

Maximum operating frequency = 1 / critical path

Shortest path : contamination delay t_{ccq} • # gates

Critical path : propagation delay t_{pcq} • # gates
longest path

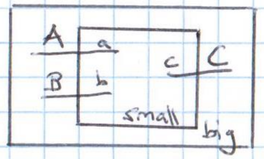
! wiring has no delay! $T_c \geq t_{pd} + t_{pcq} + t_{setup}$! Inverter!

Example:

```
module name (input [:] a, output reg [:] b, input c, output [:]d);

    wire [:] e ;                    Listable if same type and bits!
    reg  [:] f ;
    parameter g = 3'b001;
    assign e= a ;

    always @ (*)
        f <= a;

    always @ (posedge ...)
        begin
            if (c)   b <= f;
            else     b <= e;
            case (a)
                4'b0011 :   d <= 7'b001_1010;
                :           d <= {3'b..., 4'b....}
                default:    d <= 7'b0;
            endcase
        end

    assign h[0]=(a[1]) ? a[1] : a[2] ;      Stackable!
                True or False?  True  False

endmodule
```

! If there is no begin in an always block, the first instruction will be treated sequential and the following ones combinational

It is not possible to define a module in an other one




```
module big (...);
    small name1 (.a(A), .b(B), .c(C));
endmodule
```

Instruction Formats

R-Type (Register Operands)

OP	RS	RT	RD	Shamt	Funct
6	5	5	5	5	6

(add, sub, mul, div, slt, XOR, ...) add rd, rs, rt

rs, rt : Source registers
rd : destination register
op : Operation code (0 for R-type instructions)
funct : the function to perform
shamt : Shift amount for shift instruction, Ø otherwise

I-Type (Immediate Operands)

OP	RS	RT	Immediate
6	5	5	16 ← constant

imm : 16 bit two's complement

(beq, addi, subi, sw, lw, ...) addi rt, rs, imm

J-Type (jumping)

OP	addr
6	26 ← constant

(j, jr, jal, ...)

Cache

Temporal locality : load just that word into cache

spacial locality : load the whole block into cache

If a word is loaded :
1. Search cache 1 cycle
2. Search memory X cycles

Time needed with miss : 1+x

Compulsary misses : when data is loaded the first time and is not in cache
+ can be improved with larger block size

direct mapped cache : Every mem. location has only one cache location

mem. adr. $\equiv_{\text{Block Size}}$ Cache adr.

reduces conflict misses

set associative cache : Memory blocks maped to exactly one Set. In the set, the blocks can be stored in any block.

Conflict misses : Cache misses because data in cache was replaced.

Multicycled Processors

+ Area critical components are reused → reducing area
+ simple instructions finish faster
+ critical path reduced → higher frequency
- more complex control, higher control overhead
- not necessarily faster. Depends on average CPI

Data hazard: if next instruction reads result

Speed:

Latency: Time to do one complete instruction

Throughput: number of instructions per timeunit

Operating frequency: 1 Operation / 1 Time unit

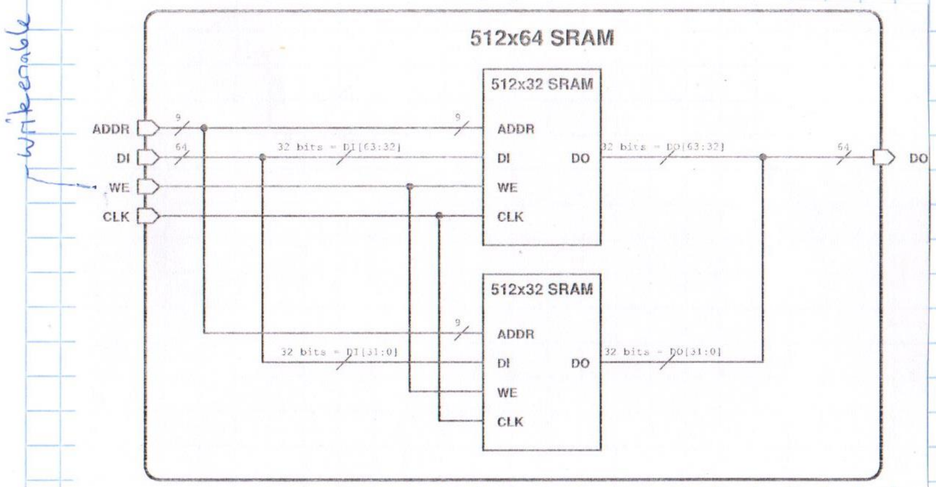
Increase speed: $T = N \cdot CPI \cdot \frac{1}{f}$

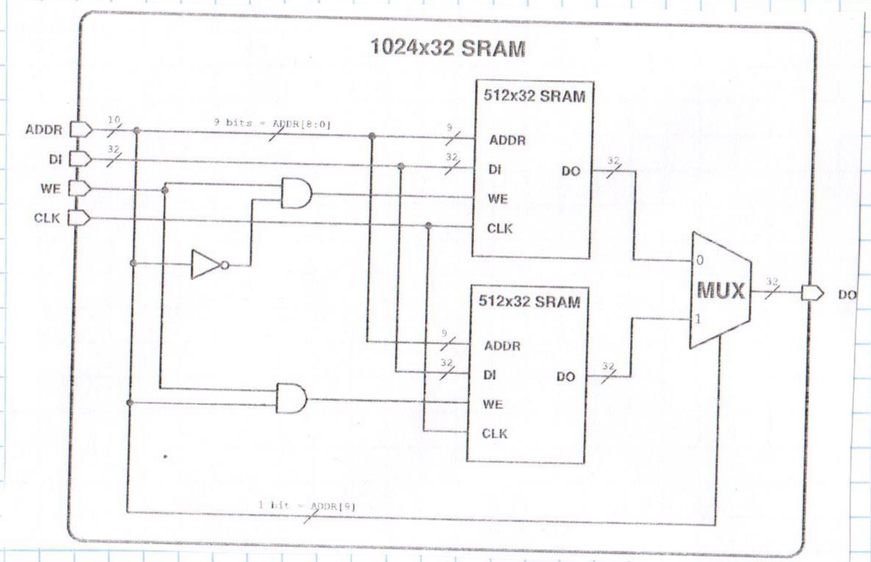
f increase
- more modern manufacturing technology
- adapt pipelining
- redesign / improve timing-critical components
- overtact

N reduction
- adapt CISC (complex instruction set computer)
  → one instruction can do more
- improve compiler → more optimized code

CPI reduction
- adapt RISC (Reduced Instruction set computer)
  → simpler instructions are executed faster
- add parallel execution units
  → do more per cycle

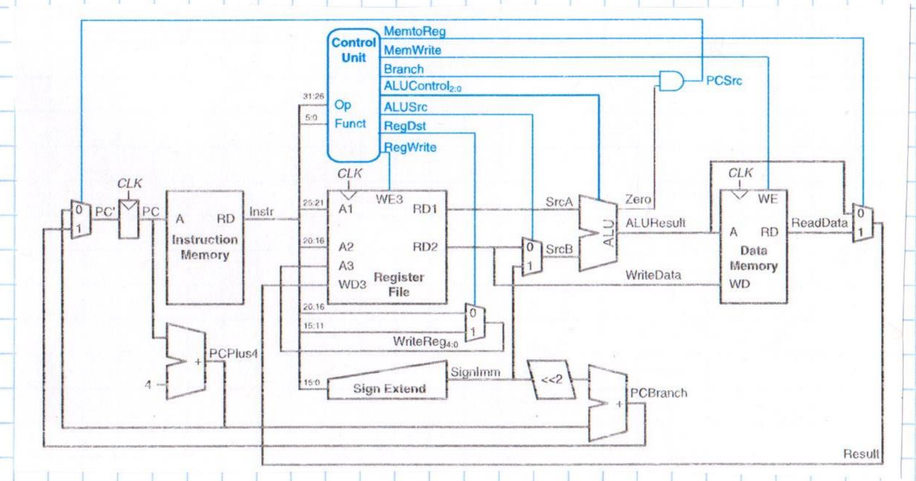
SRAM (sequential)





MIPS

Single-cycle



Multi-Cycle