MSB → 0 1 0 1 ← LSB

Unsigned Integer $[0, 2^n - 1]$

Sign/magnitude Integer $[-2^{n-1}+1, 2^{n-1}-1]$

Two's complement $[-2^{n-1}, 2^{n-1}-1]$

+0: 0000 $2^{n-1}-1$: 0111 −1: 1001
−0: 1000 $-2^{n-1}+1$: 1111 +1: 0001
0: 0000 −1: 1111 $2^{n-1}-1$: 0111
+1: 0001 −2: 1110 -2^{n-1}: 1000

Sign/Mag → Two's complement:
1. Invert all bits, besides MSB 2. Add 1_b
Twos → Sign/mag 1. Sub 1 2. Invert all bits

Sign: −5: 1101 →(1) 1010 →(2) 1011 (Two's)
−0: 1000 →(1) 1111 →(2) 0000

Sign extension (two's complement 4→8): Copy the MSB:
−5: 1011 → −5: 11111011 Negate: ~B+1

AND: • D | OR + D | XOR: ⊕ D

2 implicant

Sum-of-products:

A	B	Y	minterm
0	0	0	$\bar A \bar B = m_0$
0	1	1	$\bar A B = m_1$
1	0	0	$A \bar B = m_2$
1	1	1	$AB = m_3$

$Y = \bar A B + AB$; $Y = (A+B)(\bar A + B)$

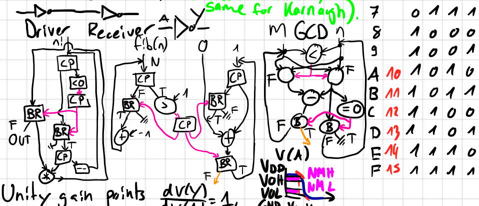
Product-of-sums:

A	B	Y	Maxterm
0	0	0	$A+B = M_0$
0	1	1	$A+\bar B = M_1$
1	0	0	$\bar A + B = M_2$
1	1	1	$\bar A + \bar B = M_3$

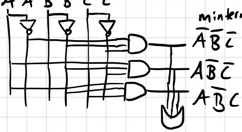
(choose 0's and invert, same for Karnaugh).

Nibbles in Base:

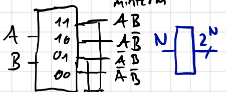
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A 10	1010
B 11	1011
C 12	1100
D 13	1101
E 14	1110
F 15	1111



Unity gain points $\frac{dV(Y)}{dV(A)} = 1$



Decoder (2^N one-hot)



Full adder: $S = A \oplus B \oplus C_{in}$, $C_{out} = AB + AC_{in} + BC_{in}$

Priority circuit: $A_3 > A_2 > A_1$: $Y_3 = A_3$, $Y_2 = \bar A_3 A_2$, $Y_1 = \bar A_3 \bar A_2 A_1$

AND = $\overline{\overline{AB}\,\overline{AB}} = \overline{\bar A + \bar A} + \overline{\bar B + \bar A}$ OR: $\overline{\overline{AA}\,\overline{BB}} = \overline{\bar A + \bar B} + \overline{\bar A + \bar B} = \overline{\bar A \bar B}$

XOR3: $\bar A \bar B C + \bar A B \bar C + A \bar B \bar C + ABC$ NOT: $\overline{AA} = \overline{A+A}$

$A < B$: $\bar A_1 B_1 + \bar A_0 B_1 B_0 + \bar A_1 \bar A_0 B_0$

$A > B$: $A_0 \bar B_1 \bar B_0 + A_1 \bar B_1 \bar B_0 + A_1 A_0 \bar B_0$

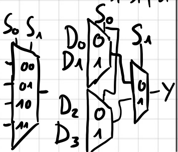
Z: Floating value/High impedance

X: Contention (Illegal value)

Multiplexer = $\bar S D_0 + S D_1$

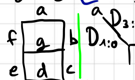
E	A	Y
0	X	Z
1	0	0
1	1	1

Tristate Buf.



	AB 00	$\bar A B$ 01	AB 11	$A\bar B$ 10
CD 0	1	0	1	1
1	1	0	0	1

$A\bar C + \bar B$



.1000 = 0.5000	.03125000
.0100 = 0.2500	.01562500
.0010 = 0.1250	.00781250
.0001 = 0.0625	.00390625

Fixed point numbers:

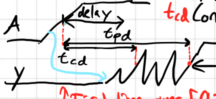
Sign/magnitude = 1010,0110

Two's compl. = 1101,1010

→: causes

t_{pd} Propagation delay: Time of crit. path

t_{cd} Contamination delay: Time of shortest path



Control-critical multiplexer: T · critical path starts from S_i

Data-critical multiplexer: The critical path starts from D_i

$T_c(s)$ = Cycle time / clock period $f_c(H_z) = \frac{1}{T_c}$ = clock frequency

Dynamic discipline: Don't change values during aperture time

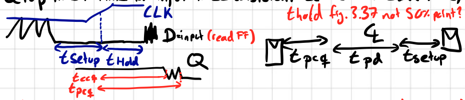
Static discipline: No values in forbidden zone ↳ Discretiz.

Sampling D: Copy input (D) to FF and output to Q after CLK.

Aperture time: $t_{SETUP} + t_{HOLD}$ Hold time = After CLK rise

Setup time: Time for input to be consistent (before 50% CLK time)

t_{hold} fig. 3.37 not 50% point?



t_{ccq} = Contamination delay of FF (from CLK ↗ to Q)

Setup time-/max-delay-constraint: $t_{pd} \le T_c - (t_{pcq} + t_{setup})$

Hold time-/min-delay-constraint: $t_{cd} \ge t_{Hold} - t_{ccq}$

Back-to-back FlipFlops ⇒ $t_{ccq} = 0$ ∧ $t_{pcq} = 0$

Metastability: Violating the aperture time.

Needs an unbounded resolution time to flip the output to 0/1

If FF input does not change resolution time = t_{pcq}

t_{skew} = Clock skew (50% CLK ↗ to 50% CLK ↘)

$t_{pd} \le T_c - (t_{pcq} + t_{setup} + t_{skew})$ || $t_{cd} \ge t_{Hold} - t_{ccq} + t_{skew}$

Glitches: Output changes a few times until correct (fix with buffers)

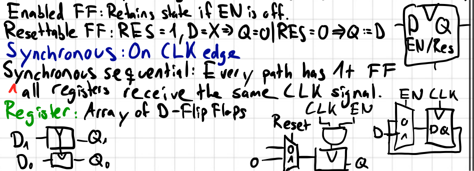
Enabled FF: Retains state if EN is off.

Resettable FF: RES = 1, D = X ⇒ Q = 0 | RES = 0 ⇒ Q := D

Synchronous: On CLK edge

Synchronous sequential: Every path has 1+ FF ∧ all registers receive the same CLK signal.

Register: Array of D-Flip Flops



Spatial parallelism: Multiple copies (N) of the circuit

Temporal parallelism: Multiple tasks across stages (S_i)

Latency = Total time; Throughput = #jobs / t

Throughput spatial = N / Lat. | Throughput temporal = $1 / \max(S_i)$

1/9.5 ns = 0.105 Ghz = 105 Mhz

Latch (Bistable)

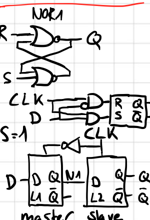
SR-Latch

R=1, S=0 ⇒ Q=0, $\bar Q$=1 | R=0, S=1 ⇒ Q=1, $\bar Q$=0

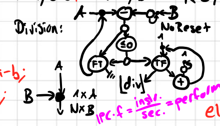
R=1, S=1 ⇒ Q=0, $\bar Q$=0 | R=0, S=0 ⇒ Q = prev. state

D-latch (data latch)

CLK=1, D=0 ⇒ R=1, S=0 | CLK=1, D=1 ⇒ R=0, S=1




```
gcd(a,b) {
  while (a≠b) {
    if(a<b) a=a-b;
    else b=b-a;
  } return a;}
```



IPC · f = instr./sec. = perform.

if(b==0) return a; else return gcd(b, a%b);

Execution time = (#instructions) (cycles / instruction) (seconds / cycle)

= CPI = $T_c = \frac{1}{f}$

Throughput = instructions / cycle = IPC ↳ 1 for SC

$2^{20} - 1 = 11...11$ (20) 20:4=5 FFFFF

2^{30} = GiB

2 bytes

$16^0 = 2^0 = 1$	$16^2 = 2^8 = 256$	$16^4 = 2^{16} = 65537$	16^1 16
$2^1 = 2$	$2^9 = 512$	$2^{17} = 131'072$	1b 16^2 256
$2^2 = 4$	$4^5 = 2^{10} = 1024$ KiB	$2^{18} = 262'144$	16^3 4'096
$2^3 = 8$	$2^{11} = 2048$	$2^{19} = 524'288$	2b 16^4 65'536
$16^1 = 2^4 = 16$	$16^3 = 2^{12} = 4096$	$16^5 = 2^{20} = 1'048'576$ MB	16^5 1'048'576
$2^5 = 32$	4^6 $2^{13} = 8192$	4^{10} $2^{21} = 2'097'152$	3b 16^6 16'777'216
$2^6 = 64$	$2^{14} = 16384$	$2^{22} = 4'194'304$	16^7 268'435'456
$2^7 = 128$	$2^{15} = 32768$	$2^{23} = 8'388'608$	4b 16^8 4'294'967'296

N bytes = 2^{8N}

Allocate s0-s7, a0-a3, ra | ++ Area, speed/throughput, power/energy, design time

factorial:

```
addi $sp, $sp, -8
sw   $a0  4($sp)
sw   $ra  0($sp)
addi $a0, $t0, 0
jal  factorial
lw   $a0  4($sp)
lw   $ra  0($sp)
addi $sp, $sp, 8
jr   $ra
```

```
module ram #(parameter N=6, M=32)(
  input clk, we, input[N-1:0] adr,
  input[M-1:0] din, output [N-1:0] dout);
  reg [M-1:0] mem [2**N-1:0];
  always @(posedge clk)
    if (we) mem[addr] <= din;
  assign dout = mem[addr];
endmodule
```

Sequential circuit := Output may depend on previous state (stored reg)

0x12348000 lui $s0, 0x1234; ori $s0, $s0, 0x8000

addiu addi only for −32768 → 32767 0x7FFF

mult $s0, $s1; mflo $a0; mfhi $a1

div $s0, $s1; mflo $a0 (quotient); mfhi $a1 (Remainder)

for(i=1; i<101; ++i) loop: slt $t1, $s0, $t0; beq $t1, $0, done

if(s<2) addi $t0, $0, 2; slt $t1, $s0, $t2; beq $t1, $0, else

Exception handling (internal) | Interrupts (external)

EPC: Adress of instr. that caused exc. | BadVAddr | Cause | Status

```
module smilingsnail (input clk, reset, a, output y);
reg [1:0] state, nextstate;
parameter S0 = 1'b0;
parameter S1 = 1'b1;
always @(posedge clk, posedge reset)
   if (reset) state <= S0;
   else state <= nextstate;
always @(*)
 case (state)
   S0: if (a) nextstate <= S0;
       else nextstate <= S1;
   S1: if (a) nextstate <= S0;
       else nextstate <= S1;
   default: nextstate <= S0;
 endcase
//output logic (mealy dep.)
assign y = (a && state == S3);
endmodule //↑ Mealy (dep. input)
```
case is a seq. statement · Use else begin/end · not else if · not in book!

```
module dlatch (input clk,
  [3:0] d, output reg [3:0] q);
always @(clk, d) //on change
  if (clk) q <= d;
endmodule

module flipflopenasyncreset(
 input clk, reset, en, [3:0] d,
 output reg [3:0] q);
 always @(posedge clk, negedge reset)
 if (reset == 'b0) q <= 0;
 else if (en & clk) q <= d;
endmodule

module tristate (input [3:0] a,
  en, output [3:0] y);
  assign y = en ? a : 4'bz;
endmodule
```

```
always @(irq) begin
{int2, int1, int0} = 3'b000;
casez (irq)
 4'b1??: int2 = 1'b1;
 4'b01?: int1 = 1'b1;
default: {int2, int1, int0} = 3'b10;
endcase
end
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```
a | b ^ c & d = a | (b ^ (c & d))

NAND: ~& NOR: ~| XNOR: ~^ (Can be vectorised)

Behavioral models: Define logic of circ. Structural models: Define interactions of instanced submodules. (Synthesizeable models & a)

//c₂ c₁ d₀ d₀ d₀ c₀ 101

Reduction operators are bitwise: &a |a ~a ^a

decoder hi #(16) (9); module decoder #(parameter N=3) (in...

Testbenches (DUT = Device under test)

```
module testbench();
reg a, b, c; wire y;
sillyfunction dut(a, b, c, y);
reg [2:0] testvectors [100000:0];
initial begin
a=0; b=0; c=0; #10; //# indicates a delay
c=1; #10; //old values will stay
if (y !== 0) $display("001 failed");
$readmemb("example.tv", testvectors);
end
always @ (posedge clk) begin
 #1; {a, b, c} = testvectors[100];
end / endmodule
```

example.tv
10_1
1_01
111
⋮

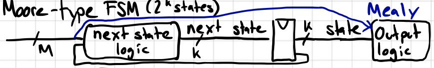
always block := write reg; assign outside := wire

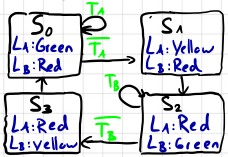
Variables modified in initial blocks must be registers (2D register arrays)

'timescale 1ns/1ps: Units are 1ns and precision of sim. is 1ps

Simplicity favors regularity = Large funct. ⇒ Multiple instr.
Make the common case fast = Small instr. set
Good design demands good compromises
Smaller is faster

Moore-type FSM (2^k states)





State transition

S	TA	TB	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

State encoding

S	S1	S0
S0	0	0
S1	0	1
S2	1	0
S3	1	1

Next state logic

Current S1	S0	Inputs TA	TB	Next S1'	S0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
⋮		⋮			

Output logic

S1	S0	LA1	LA0	LB1	LB0
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output encoding (Binary / one-hot)

green	00
yellow	01
red	10

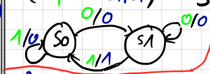
$S_1' = \overline{S_1} S_0 + S_1 \overline{S_0}\, \overline{T_B} + S_1 \overline{S_0} T_B = S_1 \oplus S_0$

Divide-by-N counter: S0 (y:1) → S1 (y:0) → ... → Sn-1 (y:0)

Moore → Mealy: Merge if nextstate is equal

Moore: Output of node is shown when moving away.
Mealy: Output of edge as soon as it moves.

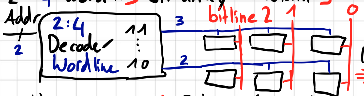
(Input/Output) Snail smiles on encountering 01

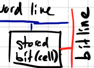


Addr	Data
11	010
10	100

depth = $2^2 = 4$, width = 3

$2^2 = 4$-word × 3-bit array





Wordline = HIGH ⇒ Read/Write

Bitline = Z ⇒ Read Bitline = (0/1) ⇒ Write 0/1

Three-ported memory: [A1] → RD1, [A2] → RD2,

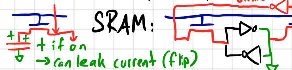
RAM (Random access memory): Volatile = Only when power

ROM (Read-only memory): Nonvolatile = Keep data

DRAM (Dynamic RAM): Store data using capacitor charge → needs refreshing, slow, 1 transistor/cell

SRAM (static RAM): Store data using cross-coupled inverter → medium, 6 trans./cell

Refresh

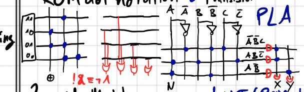


FPGA (Field programmable gate array)

Consists of: CLB (configurational logic block) = LUT + registers, Multiplexers

FF (super expensive)

ROM dot notation • Transistor



2-word-M-bit memory LUT/ROM $\log_2$

Pipeline depth = # of stages

Scoreboarding → Stall after dep. (Stall on WAW, WAR)

Potential critical path for Tomasulos algorithm

F D E E W Tag broadcast → Value capture → Instruction wakeup
F D - - D E (no forwarding, see in class^ P. Exc. 9.b)
or F D - D E (internal register forwarding)
or F D - E E (full forwarding) WAW/WAR are handled how?

In-order / out-of-order writeback | Single/Multiported memory WB

In-order dispatch = First execute must be in-order

D S - - E E (S = Issue decoded instr. into reservation station)
F D - - S E

Caller-saved registers := temporary/non-preserved registers
Callee-saved registers := Saved/preserved registers
Architectural state := PC + register set
Register set := Collectively the registers.
Register file := 2^6 = 32-word × 32-bit two read/one write port.
RISC = Reduced instruction set computer
32-bit adress with 32-bit data words byte-adressable.

Byte-adressable

Word adressable		Little Endian		Big Endian	
0001	ABCX	0004	ABCX	0004	ABCX
0000	FFAB	0000	FFAB	0000	FFAB
32-bit		4-byte = $4 \cdot 2^4 = 2^6 = 32$	03 02 01 00 / MSB LSB		00 01 02 03 / MSB LSB

sw $s0, 4($0) Effective adress: Offset + Base adress register!
lb $s0, 5($0) ⇝ C in little-endian, B big-endian
lw $s0, 4($0) ⇝ XCBA in little-endian, ABCX big-endian
lw $s0, 5($0) ILLEGAL mod 4 ≠ 0

0xcafe2b3a	0	1	2	3
Little-Endian:	3a	2b	fe	ca
Big-Endian:	ca	fe	2b	3a

Little Endian: Kleinste Stelle an kleinster Memory Adresse!

LC-3b Memory model 2^{16}; 16-bit (2 byte) adressibility, 2bit-word-align

0000	CD
0002	AB
0004	FF

LDW DR, Base, offset6 ⇔ DR = Mem[BaseR + LSHF(SEXT(offset6), 1)]

+0100 LEA ... DR = PC' + LSHF(SEXT(PCoffset9), 1)
Only loads effective adress, not memory there!

Loading: Enable/disable input to be written to a register.
Gating: Enable/disable input to bus using tristates.
General purpose registers: 8 * 16 bit registers
Condition codes: P (written reg is positive), N (negative), Z (zero)
Memory mapped I/O xFE00 - xFFFF for I/O
Setcc()

R-type (register)

31 26	25 21	20 16	15 11	10 6	5 0
op 6	rs 5	rt 5	rd 5	Shamt 5	funct 6

add, sub, and, or, slt, jr, mult, div: or rd, rs, rt
sll, srl, sra
<< >> >>>

I-type (immediate)

31 26	25 21	20 16	15 0
op 6	rs 5	rt 5	Sign Imm

lw rt ↦ 12($rs) addi rt:=rs+I ori, lui
sw rt ↤ 12($rs) beq rs, rt, done

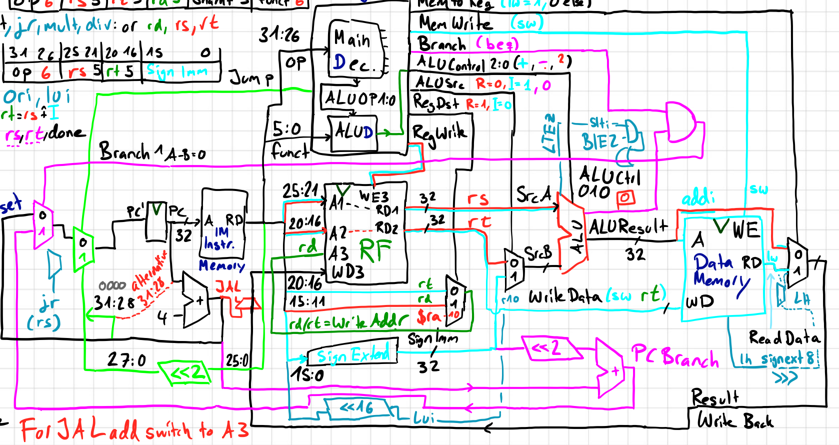
Base adressing
lw $s0 16($t0) (4x)
16-bit Sign Imm offset
beq $s1, $s2, done

Pseudo-direct adressing
j, jal .0000 X··X 00

J-type (jump)

31 26	25 0
op 6	addr 26

$PC_{31:28}$ 0000 X··X 00
j done | jal done



For JAL add switch to A3

Control = Control unit ; Program exec time = #instr. · $CP_{=1}$ · T_c

PC: 0x00400000 on reset   PC' = PC + 4 + SignImm << 2

Datapath = PC + IM + RF + DM ;

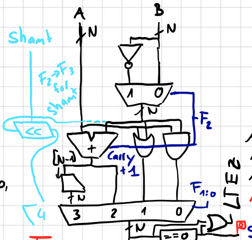
+Multicycle can re-use functional re-use

MAIN DECODER

	Opcode ⇒	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp (in dec)	
R-type	000000	1	1	0	0	1	0	10 (funct)	
lw (lh)	100011	1	0	1	0	0	1	00	+ LH
sw	101011	0	X	1	0	1	X	00	+
beq	000100	0	X	0	1	0	X	01	−
addi	001000	1	0	1	0	0	0	00	+
j	000010	0	X	X	X	0	X	XX	JUMP=1
blez	000110	0	X	0	0	0	X	01	− BLEZ
jal	000011	1	10	X	X	0	X	XX	JAL Jump
slti	001010	1	0	1	0	0	0	11 (slt)	
lui	001111	1	0	10	0	0	0	00	+

jr (R) 000000 → Modify ALU Decod. JR [JUMP]

{000,



	F
A&B	000
A|B	001
A+B	010
A+B<0	011
A&~B	100
A|~B	101
A−B	110
SLT	111
SLL	0100

ALU Decoder

ALUOp	funct		ALUControl	
00	X		010	+
01	X		110	−
10	100'000	add	010	+
10	100'010	sub	110	−
10	100'100	and	000	and
10	100'101	or	001	or
10	101'010	slt	111	slt
11	X	(slti)	111	slt
10	000:000	sll	0100	<<
10	100:001	jr	XXX	JR

STATE	IorD	RegWrite	RegDst	ALUSrcA	ALUSrcB	MemWrite	MemtoReg	IRWrite	PCSrc	PCWrite	Branch	ALUOp
Fetch	0		–	0	01		–	✓	00	✓		00 (+)
beq Decode	–		–	0	11		–		—			00 (+)
R Execute	–		–	1	00		–		—			10 (funct)
R ALUWriteBack	–	✓	0	–	—		0		—			—
lw/sw MemAddr	—		–	1	10		–		—			00 (+)
lw MemRead	1		–	–	—		–		—			—
sw MemWrite	1		–	–	—	✓	–		—			—
lw MemWriteBack	–	✓	0	–	—		1		—			—
Addi Execute	–		–	1	10		–		–			00 (+)
Addi WriteBack	–	✓	0	–	—		0		—			—
beq Branch	–		–	1	00		–		01	⇐	✓	01 (−)
j Jump	–		–	–	—		–		10	✓ ⇒		—

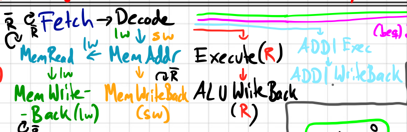
T_c (Single) $= t_{pcq\_PC} + 2t_{mem} + \max(t_{RFRead}, t_{sext}) + 2t_{mux} + t_{RFSetup}$

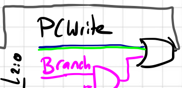
Memory read | Reg. File read | Sign ext. | multiplexer

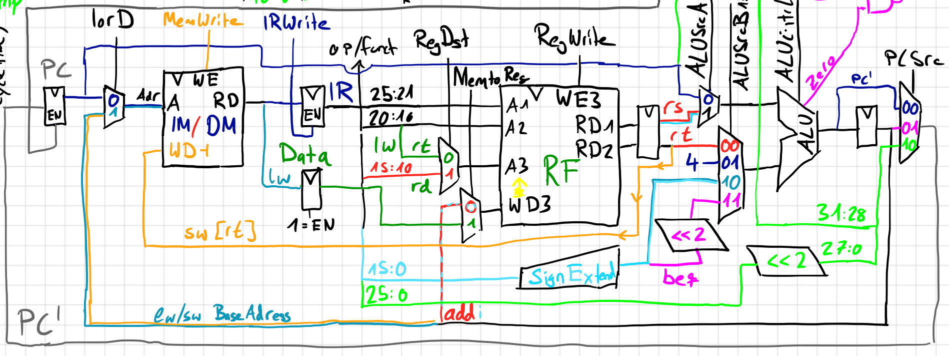
IorD Read instr. on fetch or access data memory | IRWrite Store instr. on fetch

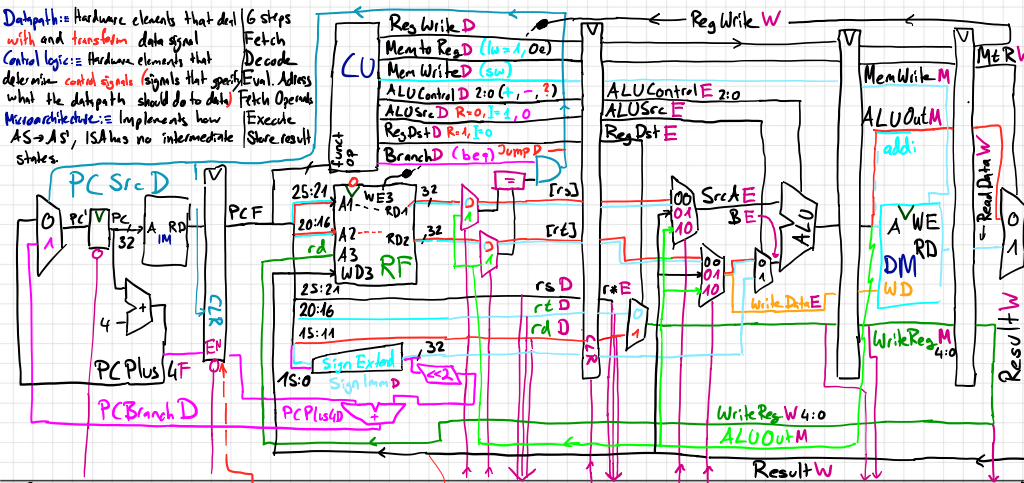
T_c (Mult.) $= t_{pcq} + t_{mux} + \max(t_{ALU}, t_{mux}, t_{mem}) + t_{setup}$

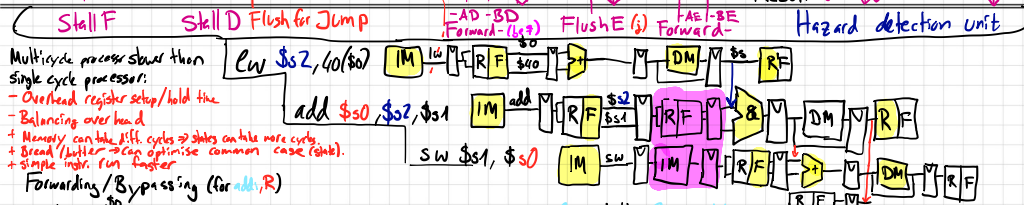
Register CLK to Q | Memory Read | Register setup time











Multicycle processor slower than single cycle processor:
- Overhead register setup/hold time
- Balancing over head
+ Memory can take diff. cycles ⇒ stalls can take more cycles
+ Branch/Jump ⇒ can optimise common case (stalls).
+ simple instr. run faster

Forwarding/Bypassing (for add, R)

if((rsE != 0) ∧ (rsE == WriteReg M) ∧ RegWrite M) ForwardAE = 00;
elif((rsE != 0) ∧ (rsE == WriteReg W) ∧ RegWrite W) ForwardAE = 01;
else ForwardAE = 10

lw stall - 2 cycle latency. Stalls Fetch, Decode, Flushes Exec

lwstall = ((rsD == rtE) ∨ (rtD == rsE)) ∧ MemtoReg E
branchstall = (BranchD ∧ RegWrite E ∧ WriteReg E ∈ {rsD, rtD}) ∨ (BranchD ∧ MemtoReg M ∧ WriteReg M ∈ {rsD, rtD})

Control Hazard/Dep: Instr. to fetch unknown (beq) (if beq depends on lw)
Sol A. Stall fetch, decode, execute

StallF = StallD = FlushE = lwstall ∨ branchstall (∨ jump)

Branch prediction penalty = #flushes
Sol B. ⇒ If branch simply flush F, D, E
Sol C: Decrease branch prediction pen. by detecting in Decode using = for beq
Forward AD like Forward E ⇒ above branch

Resource contention:= Memory or functional unit needed at the same time. Structural hazard := No more available FUs

Flow dependency (RAW)
Output dependency (WAW)
Anti dependency (WAR)
False dep. eliminated using ROBs.

$$T_c = \max \begin{cases} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{cases}$$

Fe De Ex Me WB

Handle flow dependencies: Detect & wait (stalling), Detect & Bypass (Forwarding), Detect & Eliminate dep. on software level (inserting NOP)
- Predict value ∧ verify (flush) - Do something else (Threading) - Out-of-order execution (Execute later ind. code in same thread)
Inserting NOPs = ↗ #instructions, Inserting bubbles = ↗ CPI

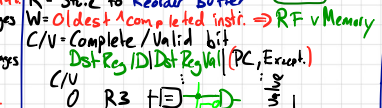
Control dependency handling
A. Solve by stalling/flushing
B. Early branch resolution
MIPS uses = for beq in Decode
- Introduces another data dep.
C. Dynamic branch resolution
2-bit branch predictor

⊖ Cons Pipelining:
- Resource contention/Structural haz
- External fragmentation (some stages not required for all instr.)
- Internal fragmentation (Not all stages have the same speed)
- Stalls add HW, PC/CP code dep.

Dependence detection mechanism
Scoreboarding (Valid bit on each reg)
+ Simple - Stall not only RAW
Combinational dep. check logic
+ Fast (only RAW)
- More complicated esp. on Superscalar machines.

Reorder Buffer (ROB)

+ Does renaming ⇒ removes false deps.
- Content addressable memory (CAM)



R = Store to Reorder Buffer
W = Oldest ∧ completed instr. ⇒ RF ∨ Memory
C/V = Complete/Valid bit
Dst Reg ID | Dst Reg Val | (PC, Except.)

+ CAM faster - Complex

ROB ≥ #registers + max stage cont = max size of instr. window.

Indirection (Alternative impl.)

1. Access RF, if valid bit = 1 ⇒ OK, else load/overwrite pointer of where it will land in ROB.
2. Check if ROB is valid, else stall and wait.
   - Increases latency (for ROB 106)
3. Finished computation, broadcast ROB 106.

+ Simple - Adds another stage (incr. latency)

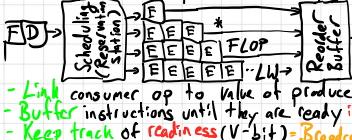
Memory dependencies are dynamic
MOB is equivalent concept to ROB.
- Huge, cannot solve all deps.
- Additionally needs to know size of access.

Data dep./hazards
lw $t0, 40($s0)
add $t1, $t0, $s1 RAW (flow dep.)
sub $t0, $s2, $s3 WAR (anti dep.)
add $t0, $s4, $s5 WAW (output dep.)
The last instr. can be squashed.

Instruction window:=
All decoded but not yet retired instr.

Out-of-order execution (Data-flow order)

Dep. instr ⇝ moved away into reservation stations



- Link consumer op to value of producer
- Buffer instructions until they are ready in reservation station
- Keep track of readiness (V-bit) ⇒ Broadcasting (compare)
- Dispatch to functional unit (FU) ⇒ Wakeup and select instr

#reserv. station entries = (usually k·#FU)

Tag size = $\lceil \log_2(\#\text{reserv. station entries}) \rceil$ bit

#tag comparators = #reserv. station entries * #source regs. * #FU + #FU * #regs.

RAT size

Re-Order (makes it Von-Neumann, used for except.)

After execution update the RAT (frontend register file)

After retirement update the architectural reg. file

= oldest instr. in machine ^executed ⇒ Updated in program order

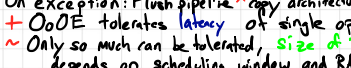
On exception: Flush pipeline ^copy architectural RF to frontend RF

+ OoOE tolerates latency of single ops.

~ Only so much can be tolerated, Size of instr. window depends on scheduling window and RAT/ROB size.

Data flow at ISA level = unsuccessful

+ Good for irregular parallelism (Need to spot dependencies).
+ Removed PC (instr. counter) dependency
- Bad for debugging ⇒ Not just sequential state.



Superscalar (N-wide ideally IPC = N; CPI = 1/N)

+ Higher IPC - Higher complexity dep. handling - More hardw.

VLIW (Very long instruction word) compilers job!

Bundles a lot of indep. instr. in one. ⇒ Moves in lock step.

+ Simple (no sched./no dep. checking ⇝ no renaming)
- Compiler needs to find N indep. instr. Problem
- Code size increases (NOP) ↑ Lock step problem

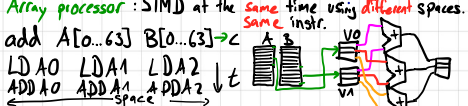
⇒ Lower I-cache hit-rate/Higher fetch memory bandwidth

- Change width ⇒ Recompile (change of ISA)
+ Compilers for VLIW can be used for superscalar

SIMD processing (Single instruction multiple data)

For e.g. dot product on 2 vectors. Modern = Array- & Vector proc.

Array processor: SIMD at the same time using different spaces.

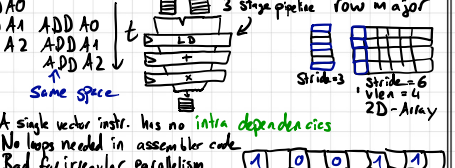


- Replicates a separate functional unit for each vector entry.

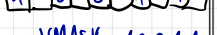
+ Fast, does VLEN instr. at same time, no dep.:

Array width / Vector length

Vector processor: SIMD at different times using same space.



+ A single vector instr. has no intra dependencies
+ No loops needed in assembler code
- Bad for irregular parallelism
- Slow for single-port memory



Memory banks VLEN register

Word-interleaving = Row buffer conf.

N-parallel accesses for stride = 1 (relatively prime to # banks)

~~If VLEN>32 should use 64 banks~~

If load latency is 50 cycles should use 64 banks.

- Row buffer conflicts at single bank ⇒ single value

↳ Resolve with more banks, random addressing, better mem scheduling

VMASK = 1011
VSTRIDE = 2

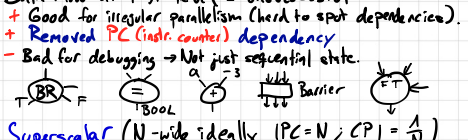
Dynamic instructions = # instr. executed by machine.

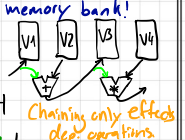
Static instructions = # instr. written down in program

Vector chaining (cannot chain loads/stores to the same memory bank!

VLEN = 32+8 = 40

Chaining & Banking B=32 = Bank size 2-ported



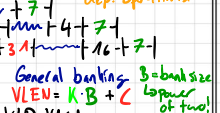


Banking, NO chaining

General banking B = bank size (power of two!)

VLEN = K·B + C

VLD V1 ← A



Vector stripmining = Split huge data into multiple vectors

Non-strided (irregular) load:

A: Gather/Scatter operations (good if sparse)

Use index vector instead of hundreds of loads:

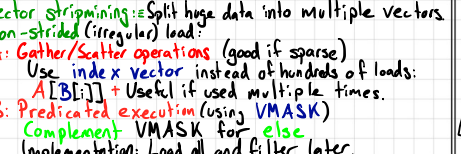
A[B[i]] + Useful if used multiple times.

B: Predicated execution (using VMASK)

Complement VMASK for else

Implementation: Load all and filter later.

Density-time: Only load masked.



Flynn's taxonomy

SISD (Scalar sequential): Pipeline / Superscalar / OoO / VLIW

SIMD (Data parallel): Array-/Vector processors

MISD: Systolic arrays/streaming processors MIMD: Multicore Superscalar

GPU (SIMT = Single instr., multiple threads)

Warp: Set of threads that exec. same instr. but with diff. data

SIMD	vs.	SIMT
[VLD, VADD, VST], VLEN		[LD, ADD, ST], # of threads
↕ vector instr.		↕ scalar instr.
Sequential instr. on ISA level		Threads are grouped dynamically

+ Can treat each thread seperately

⇒ Can be executed on a sequential machine if desired.

⇒ leads to MIMD processing

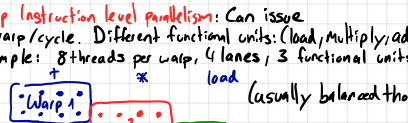
+ Granularity on the level of warps

FGMT: SIMD Pipeline (only one warp at a time)

	RF1	RF2	RF3
Contain registers of thread IDs:	1, 4, 7, 10	2, 5, 8, 11	3, 6, 9, 12

Warp instruction level parallelism: Can issue 1 warp/cycle. Different functional units: (load, Multiply, add)

Example: 8 threads per warp, 4 lanes, 3 functional units



Assumed Answers must be w/o branch divergence!

Branch divergence

Every thread can take another control flow path, warps are hypothetically regrouped on every instr., Spaces are NOP'd.

↳ actually only after branch divergence.

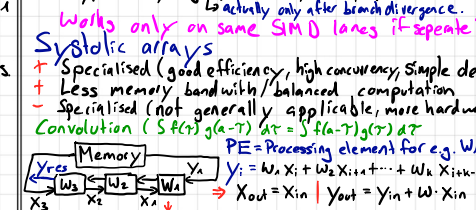
Works only on same SIMD lanes if seperate RFs.

Systolic arrays

+ Specialised (good efficiency, high concurrency, simple design)
+ Less memory bandwidth / balanced computation
- Specialised (not generally applicable, more hardware)

Convolution $(f*g)(a) = \int f(\tau)\, g(a-\tau)\, d\tau$

PE = Processing element (e.g. WARP)



Decoupled Access/Execute Processors (DAE)

Fetch: Split into A-instr. and E-instr. part

Write/Read memory done over AEQ | EAQ

Also 2 RFs which can be interchanged with Queues

+ Queues lower needed number of registers.
+ Access/Execute can run ahead of each other.
+ Easier than Tomasulo - Needs compiler support
- Branches require synchronization of A/E.
- 2 instr. streams (could be done in 1 though).

Loop Unrolling + Good if you have bad loop prediction.

+ Enlarges basic block ⇒ Enables code optimisation/scheduling
- Needs more code, also check if not multiple of unroll factor

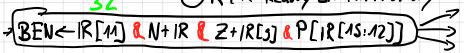
LC-3b

Critical path: S stages + 2 mem. stages + mem. latency v. control path

Pseudoinstructions → (Assembler) → Instructions → (Machine) → Microinstructions

16, 16 — 33 — 35

32



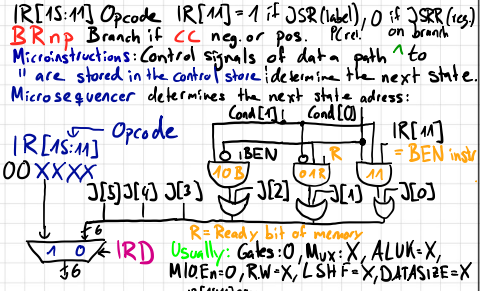
IR[15:11] Opcode IR[11] = 1 if JSR(label), 0 if JSRR(reg.)

BRnp Branch if CC neg. or pos. PCrel. on branch

Microinstructions: Control signals of data path ^ to

" are stored in the control store ↓ determine the next state.

Microsequencer determines the next state adress:



Single-bus design: + Low hardware − Reduced concurrency

S(M) Utilization = $\frac{\text{Useful ops of a warp}}{\text{Total \# of possible ops of a warp}}$

Warp: hat 32 threads, code:

if(A[i] > 0) { 4 ops }

Entweder $\frac{32}{32}$ / $\frac{160}{160}$ / $\frac{32+4\cdot a}{160} = \frac{4\cdot(8+a)}{160} = \frac{8+a}{40}$

$\frac{8+a}{40} = \frac{1}{5} \Rightarrow 8+a = 8 \Rightarrow a = 0$ & $a \in [1;31] \Rightarrow$ Impossible

a in every 32 of B's elements is less than 0

Weekly warp ≥ 1/5

LC-3b

+ Simple design for powerful computation (hidden μ-Ist)
+ Easy extensibility of the ISA
+ Patchable Microcode

Technology	Cost/GB	Access Time
SRAM	~10'000 $	~1 ns
DRAM	~100 $	~100 ns
Hard Disk	~1 $	~10'000'000 ns

Cache, TLB

MM = Physical memory, Page T. Root of virtual memory

Miss rate = # Misses / # memory accesses = 1 − Hitrate

Hit rate = # Hits / # memory accesses = 1 − Missrate

AMAT (Average Mem. Access time):

$t_{cache} + MR_{cache}(t_{MM} + MR_{MM} \cdot t_{VM})$

Spatial locality: When accessing data, bring data nearby.

Temporal locality: Keep recently accessed data in higher levels of cache.

C = Capacity (# of bytes)

b = Block size (bytes of data brought in at once)

B = C/b = # Blocks in cache ≈ # Cache lines

N = # of blocks in a set (= Degree of associativity)

S = B/N = # of sets ≈ # of cache rows

2^i

Tag store Size N-way cache: $S \cdot (N \cdot (t + 1 + 1) + (\log_2(N))(N-1))$ (tag size | valid | dirty bit | LRU)

Data store size N-way cache: $S \cdot N \cdot b$ (bytes)

Compulsary miss: First time accessing this particular memory.

Conflict miss: Won't happen in a fully assoc. cache with LRU rep.

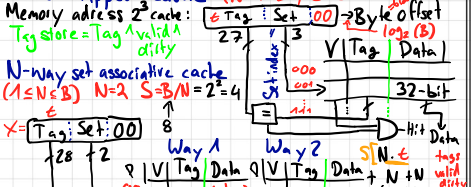
Capacity miss (eviction): Even happen in fully assoc. cache with LRU rep.

LRU replacement = Oldest / MRU replacement = Newest

Direct mapped cache (N=1=L, S=B=2^3)

Memory adress 2^3 cache: t | Tag | Set | 00 → Byte offset (Block offset) $\log_2(B)$

Tag store = Tag ^ valid ^ dirty



N-way set associative cache (1 ≤ N ≤ B) N=2 S=B/N=2^2=4

if [(TagWay1 == Tag && Valid of Way1) || (TagWay2 == Tag && Valid)]

Additionally needs a multiplexer to choose data!

Fully associative cache (N=B, S=1)

Usually small due to the many comparisons!

Example: Direct mapped cache Block size = 2^2 = 4 (offset)

S = 2^1 = 2 = B

0 2 4 8 16 32 has 2 cache hits with Block size 8

0 8 16 24 16 8 0

0 0 1 1 2 0 0

X ↦ ⌊X/B⌋ % S

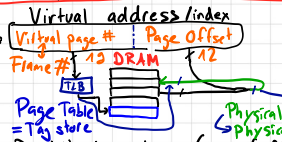
↳ Cache row = cache set

B = 16/S = 2

1 page table / process

temporal locality ↗ = set # ↗

Spatial locality ↗ = block size ↗ (# of diff. indep. mem. = N (degree of ass.))



2^{12} byte Page ~ 4 KB

2^{32} bit - 16 byte Page

Physical (logical) address

Page table links chunks (size of offset) 0-4096 → 4096-8191

DRAM: Frame # | Frame content — Check TLB → Cache hit

LRU-Policy (Uniprocessor):

0	Page 13
1	∅
2	Page table

Else Check page table for valid bit

Not valid → Page fault

Translation Lookaside Buffer (TLB) = fully associative