Computer Architecture

Performance

- X is n times faster than Y

$$\frac{Perf(x)}{Perf(y)} = \frac{Execution\ time(y)}{E(x)} = n$$

- CPU time (sec) = $\frac{\#\ cycles/program}{frequency\ (Hz)}$ ($\frac{cycle}{sec}$)
- $F_i = \frac{\#\ i\ type\ instr}{prog\ instr\ count}$
- avg cycle/instr $CPI = \sum_{i=1}^{n} CPI_i \cdot F_i$
- CPU execution time (sec/program) = $\frac{instr\ count}{\#\ cores}$ (avg CPI) $\frac{1}{frequency\ (Hz)}$
- MIPS = $\frac{inst\ count}{execution\ time \times 10^6}$
- benchmark → standardized tests

CPU Power

- $P = VI$
  = (capacitative load)(volt)^2(freq)
- faster CPU = more heat
  - ↳ reduce heat by (cooling, ↓voltage, ↓leaked current, ↓clock speed Yo ↓performance)

Instructions

- → R type - deals w/ values in registers
- → I type - memory access (load)
- → S type - memory access (store)
- → SB type - branch instrs (2^{12} range)
- → U type - upper immediates
- → UJ type - jump instrs
- ←See "Commands" for formats

Registers

- x0 - constant 0
- x1 - return address
- x2 - stack pointer
- x3 - global pointer
- x4 - thread pointer
- x5-x7, x28-x31 - temporary register
- x8 - frame pointer
- x9, x18-x27 - saved register
- x10-x17 - function argument/result

Addressing

Low Addr

	0	1	2	3	4	5	6	7	MSB
Byte	0	1	2	3	4	5	6	7	← Little endian
Byte	7	6	5	4	3	2	1	0	Big endian

78 a1 b2 c3 d4 e5 f6 g7 h8

Little → h8 g7 f6 e5 d4 c3 b2 a1

Big → a1 b2 c3 d4 e5 f6 g7 h8

- Arrays
  - ↳ A[i] → base + offset
    offset = (type size)(index)
  - ↳ ie A[4] 8 bit → (8)(4) = 32
    32+ base addr = A[4]
- can slli to adjust for type size

Machine Language ⟷ RISCV

31 25	24 20	19 15	14 12	11 8	7 6 0	
funct7	rs2	rs1	fn3	rd	opcode	R
imm[11:0]		rs1	f3	rd	opcode	I
[11:5]	rs2	rs1	f3	[4:0]	opcode	S
[12]	[10:5]	rs2	rs1	f3	[4:1] [11]	opcode	B
imm[31:12]				rd	opcode	U
[20]	[10:1]	[11]	[19:12]	rd	opcode	J

- Lui tip
  - ↳ load 0xDEADBEEF to x10
    Lui x10, 0xDEADC
    +1 b/c not value B = 1011 starts w/ 1
    addi x10, EEF
    → b/c Lui writes to 31:12 sign extends 63:32 & clears 11:0

Control Flow

- ↳ decision making instrs

Ex Loop:

```
<Procedure>
bne x#, x#, Exit
<Procedure if not Exit>
beq x0, x0, Loop
Exit: ...
```

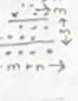
guarantees no error thrown (may cause ∞ Loop though)

- brach instr are SB type w/ range 2^{12} bits
- location addr's must be even #s if given odd
  ① subtract 1
  ② raise exception/error
- for moving btwn instrs
  - ↳ jal jump & link, go to procedure 2^{16} bit reach
  - ↳ jalr jump & link register; return to previous branch
- when jumping procedures
  ① Store params in stack (increment stack pointer)
  ② transfer to procedure + preform ops
  ③ update values in stack for caller to use
  ④ return to place of call (jalr)

— Ch 3 & Ch 4 —

Multiplication

- → m bits × n bits = max bit product



- → Algo 1

0's (same # as multiplier)	multiplicand

Each stage
- ② if LSB of multiplier = 1
  product + multiplicand
  if Ø → do nothing
- ③ shift multiplier right
- ④ shift multiplicand left

Cont to repeat until multiplier gone

Ex $1001_2 \times 1001_2$

	multiplier	multiplicand	product
0	1001 →	00001001	00000000
1	0100	00010010	00001001
2	0010	00100100	00001001
3	0001	01001000	00001001
4	0000	10010000	01010001 ans

- max value → 64 bit × 64 bit
- ~1 cycle/step) (3 steps) (64 bits) ≈ 200 clocks

→ Optimized Algo ←

+1 bit for overflow

① add multiplier to product
Each stage
② if LSB of product = 1, add multiplicand to MSB
if = Ø → do nothing
③ shift product right
Cont to repeat until multiplier gone

Ex $1001_2 \times 1001_2$

0	1001	00001001	
1	1001	+10010011	
		01001001	
2	↓	00100100	
3		00010011	4 + 01010001 ans

Booth's Algo (Mog(64))

↳ signed multiplication
- ↳ make both #'s +,
- ↳ 2's comp any neg

* sign extend

① Partition
② if 01 → add multiplicand
if 10 → sub multipli
③ shift right

Ex 2×-3 ("mythical bit")

multiplicand 2 = 0010 | product
2's comp = 1110

1.	0000 1101 0	sub
	1110 1101	
2.	1111 0110 1	add
	0001 0110	
3.	0000 1011 0	sub
	1110 1011	
4	1111 0101 1	none

ans → 5 1111 1010

bc -3 → 2's comp ↓ = 0000 0110 ✓

Division

- stored | high - quo | low - rem |
- # rounds = max(divisor, dividend) + 1
- ① place divisor in MSB of divisor
- ② place dividend in LSB of rem

Each stage
- ③ rem = rem - div
- ④ rem ≥ 0 shift quot left & RMB = 1
  rem < 0
  rem = rem + div (restore)
  shift quot left & RMB = 0
- ⑤ shift divisor right

Cont to repeat until divisor in LSB half

Ex $00000111_2 \div 0010$

0.	0000	00100000	00000111
1.	0000	00010000	③ 00000111
			④ 00000111
2.	0000	00001000	⑤ 00000111
3.	0000	00000100	⑤ 00000111
4.	0001	00000010	⑤ 00000011
			⑤ 00000001
5.			④ 00000001

Ans → 0011 | 00000001

↳ quo 3 w/ rem 1

→ Optimized Algo ←

① place dividend in LSB of return reg
② shift return left
Each stage
③ rem = rem - div (subtract divisor from left ½ of dividend)
④ rem ≥ 0 shift return reg left rem < 0 ↳ fill w/ 1 rem = rem + div (return to b4) shift return reg left & fill w/ 0

Cont to repeat until right ½ of og return reg on left

Ex 0006 0111 ÷ 0010

0.	0010	00000111
		00001110
1.		11101110
		00011100
2.		00111100
3.		00111000
		00110001
4.		00010001
		00100011

Shift back (left ½)

00010011
rem ↑ ↑ quotient

Floating Pt

$(1.xx) \times 2^{xx}$ — significand, base, exponent/range; accuracy/fraction

In N bits, range
- → unsigned 0 to 2^N
- → 2's comp -2^{N-1} to $2^{N-1}-1$
- → signed $\pm 2^{63}$

Floating pt larger range
- → half (16 bit) 2^{-14} to $(2-2^{-10})2^{15}$
- → single (32) 2^{-126} to 2^{128}
- → double (64) 2^{-1022} to 2^{1024}

MSB	sign	exponent	fraction	
16 =	1	5	10	half
32 =	1	8	23	single
64 =	1	11	52	double

↳ $x = (-1)^{sign}(1 + frac)(2^{exp - bias})$

bias → 15, 127, 1023, $= 2^{(\#\ exp\ bits - 1)} - 1$

- denormal → hidden bit = 0, $0.xx \times 2^{(-1)^{sign}(0+\#) \to -1}$
- ± ∞ => exp = 11...11 frac = 00...00
- NAN => exp = 11...11 frac ≠ 00...00

Bin → dec
1 10000011 01110000...
$(-1)^1(1 + (2^{-1} + 2^{-2} + 2^{-3})) \times 2^{(2^7 - 2^1 + 2^0 - 127)}$
$= -30$

Normalize $10_{10} \to 1010 = 1.010 \times 2^3$

Dec → bin
Sign → 1
Exp → 3 + bias = $130_{10} = 10000010_2$
Frac → part after decimal 010
↳ 1 10000010 01000000...
Smallest exp 00..001 = 1
largest exp 11...110 value dependant type

Dec → bin -0.75
Sign → 1
Exp → -1 + bias = $126_{10} = 01111110_2$
Frac → 100...00
↳ 1 01111110 1000...

OR

		msb
0.75 × 2 = 1.5	1	↓
0.5 × 2 = 1	1	} 0.11000...
0 × 2 = 0	0	= 1.1×2^{-1} ✓

→ In RISCV ←
- only operate in floatng pt regs
- hold 32 bits
  - ↳ flw, fld, fsw, fsd, add .s/.d
  - ↳ fadd, fsub, fmul, fdiv, feq, fle, fsgnj, fld
- → Arithmetic ←
  - ① Shift smaller # to match larger & truncate
  - ② Add (2's comp - numbers)
  - ③ Normalize
  - ④ Truncate/round → 3 decimal

Ex $9.999 \times 10^1 + 0.161 \times 10^0$
① → $99.99 > 0.161$
$9.161 \times 10^0 = 0.0161 \times 10^1$
② 9.999×10^1
$+0.016 \times 10^1$ → store in overflow
10.015×10^1
③ 1.0015×10^2
④ 1.002×10^2

Multiplication
- ① Add exponents & subtract bias
- ② Multiply
- ③ Normalize
- ④ Truncate/round

Ex $(1.000_2 \times 2^{-1})(-1.110_2 \times 2^{-2})$
G(-1+127) + (-2+127) - 127 = 124
① no over/underflow
② $1.000 \times -1.110 = 1.1102 \times 2^{-3}$
③ already normalized
④ og (+)(-) = −
Ans: -1.110×2^{-3}

Rounding
↳ guard, round & sticky

Ex $(5.01 \times 10^{-1}) + (1.34 \times 10^2)$ 10^2 000 5=1

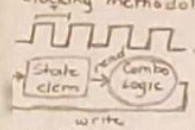
GR	GRS
5.0100×10^{-1}	5.0100×10^{-1}
$0.0050\ 00 \times 10^2$	$0.0050\ 100 \times 10^2$
$+1.34 \times 10^2$	$+1.34 \times 10^2$
1.3450×10^2	1.3450×10^2
1.34×10^2	1.35×10^2 round
↔	

Processors

- 3 stages
  1. PC supplies instr addr & fetches instr from memory (& update PC)
  2. decode instr & read regs
  3. execute instrs

* All instr except j use ALU

- 2 functional units
  1. Combinational → operate on data
  2. Sequential → contain state
- Clocking methodology

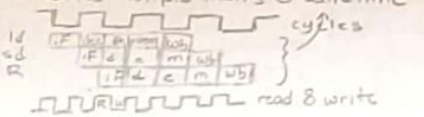


- cycle length = longest logic gate delay
- instr fetch ① read PC addr
  ② pull from instr & execute
  ③ PC += 4
- R format instr ① read 2 reg ops
  ② do arithmetic/logic ops
  ③ (Load) read mem & update reg
  (store) write reg value to mem
- branch instr ① read reg ops
  ② compare ops
  ③ calculate target addr
  sign extend / shift / PC+4
- control unit → regulates data transfer
  decodes instrs (w/ instr type & func)
  ALU input → A' + B' + OP

0	0	0 0	and
0	0	0 1	or
0	0	1 0	add
0	1	1 0	sub
0	1	1 1	set less than
1	1	0 0	nor

Pipelining

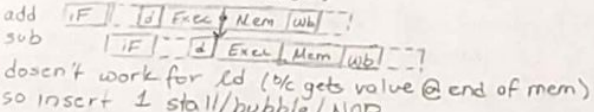
- 5 stages
  1. Fetch (instr) & update PC
  2. dec - reg read & instr decode
  3. exec - calculate mem addr
  4. mem - read data from mem
  5. wb - write back data to reg
- increases efficiency
- executes multiple instrs @ same time



Hazards

↳ situations preventing starting the next instructions

- Control Hazard
  ↳ deciding cntrl action depends on a prev instruction
  → SLTN: branch prediction
  ↳ execute a possible branch & only stall if guess is wrong
  → static → uses typical behavior
  → dynamic → table of previous decisions
- Structure Hazards
  ↳ required resource is busy, really only an issue w/ instr & data
  → SLTN: have seperate instr & data memory paths
- Data Hazards
  ↳ need data update from a prev instr
  → SLTN: forwarding (pull value after exec phase/ dont wait for write back)



doesn't work for lw (v gets value @ end of mem)
so insert 1 stall/bubble/NOP

- Can also reorder code to minimize hazards