Computer Architecture

Author: Jose 胡冠洲 @ ShanghaiTech

Full-ver. Cheatsheet

See below (page 2-7).

Green Cards

MIPS Green Card

See below (page 8-9).

RISC-V Green Card

See below (page 10-11).

Links

- Berkeley CS61C "Great Ideas in Computer Architecture" Course Website
- RISC-V Web Simulator: Venus
- MIPS Full IDE + Simulator: MARS

6 Great ideas in CA

- #1: Abstractions: Layers of Representation & Interpretation
- #2: Moore's Law: 2x Transistors / chip each year.
- #3: Principle of Locality: Memory Hierarchy
- #4: Parallelism: Amdahl's Law

  $$\text{Speedup} = \frac{1}{(1-F) + \frac{F}{\#\text{cores}}}$$, F is % can be sped-up

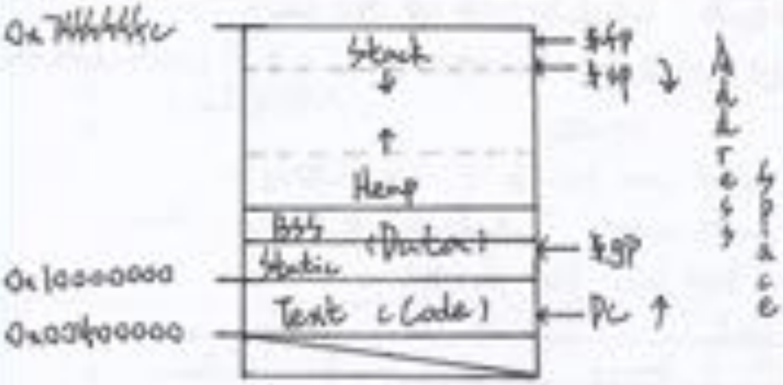
- #5: Performance Measurement & Improvement
- #6: Dependability via Redundancy.

MSB $\underset{31}{XXX}$ - - - - - - - - - - - $\underset{0}{XXX}$ LSB

Unsigned: $0 \sim 2^{32}-1$, Signed: $-2^{31} \sim 2^{31}-1$

Two's complement (补码): 各位取反，再+1.

One's complement (反码): 各位取反；两个0.



C: No deepcopy for `struct`s.

Static variables vs. On-stack (local)

*p++: * priority > ++

void* 不能解引用 / + - !

{ sizeof (int* Type) = 8 = sizeof all ptrs.
{ sizeof (int arr[10]) = 10 × sizeof (int).

{ char * s = "..." ⇒ In static section
{ char s[] = "..." ⇒ On function stack

求 remainder of 16: bit-wise & 15

(ISA) Instruction Set Architecture

(RISC) Reduced Instruction Set Computing

MIPS: 32 Registers. Each 32-bits wide

{ add: cause overflow detection.
  Carry into MSB ≠ Carry out of MSB
{ addu: No overflow detection

{ logical shift: 添0, $\times 2^n$ ⇔ sll n
{ arithmetic shift: 添 sign-bit, ÷2 ⇔ sra 1

$at: For Assembler only, ✕ user

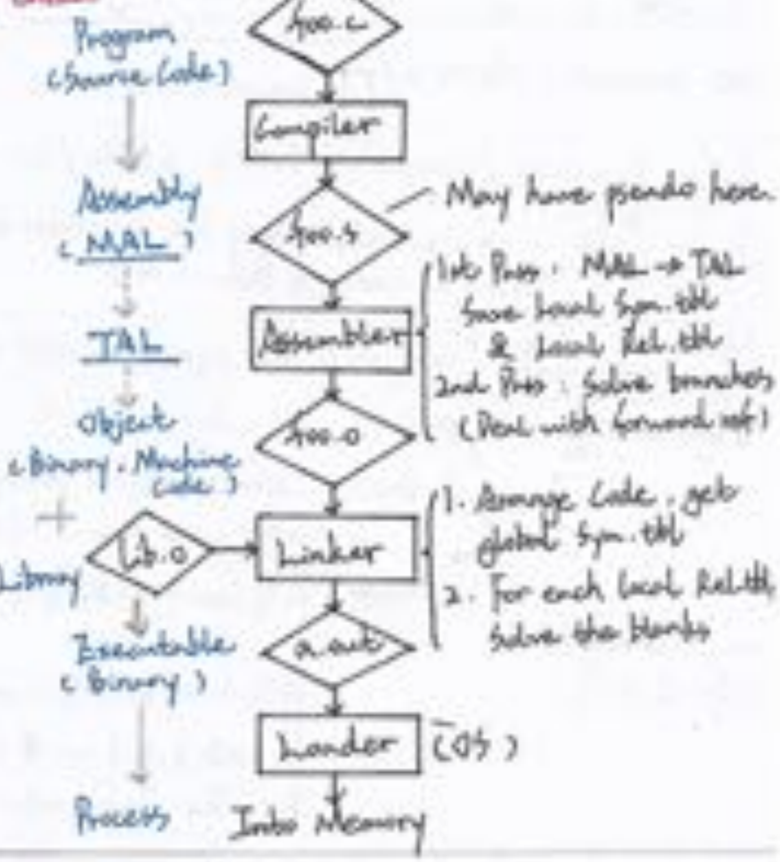
Preserved for Function calls Convention:

$sp, $fp, $gp, $s0 ~ $s7

Absolute Address Format:

{(PC+4)[31:28], Imm26, 00} ⟶ In all 32 bits represented

"CALL"



PC
- Normally +4
- On branch: PC = (PC+4) + (Imm×4)

  $$\text{Imm} = \frac{\text{Label addr.} - \text{Addr} - 4}{4}$$

- On Jump: absolute value as above

Flynn's Taxonomy

		Data: Single	Data: Multiple
Instructions	Single	SISD MIPS	SIMD Intel SSEs
	Multiple	MISD —	MIMD Multicore

Cache Replacing: LRU vs. MRU

Block size ↑
- [Compulsory ↓, Conflict ↑, Capacity →]
- [Hit time slightly ↓, Miss rate ↓ then ↑, Miss penalty ↑]

Associativity ↑
- [Compulsory →, Conflict ↓, Capacity →]
- [Hit time ↑, Miss rate ↓, Miss penalty →]

Cache Capacity ↑
- [Compulsory ↑, Conflict ↓?, Capacity ↓]
- [Hit time ↑, Miss rate ↓, Miss penalty ↑]

Performance Affect

	# of Insts.	CPI	clk rate
Algorithm	✓	✓	
Language	✓	✓	
Compiler	✓	✓	
ISA	✓	✓	✓

Use Geometric Mean: $\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$

Stalling (Assume × forwarding)

Case 1, write Reg used right after
(Assume W, D done in one cycle)
则对齐 W ↓ D by 重复这一条的D

Case 2, after branch
(Assume Equal get in stage D)
则对齐 D ↘ F by 重复这一条的F

Error Detection / Correction Codes (EDC/ECC): Hamming Distance = # of different bits

Parity Code ⟶ Generate parity bit to make sum even ⟶ Transmit ⟶ Check
Maximum Hamming Distance = 2

Hamming Code ⟶ (1 bit correction, 2 bits detection)

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011
P_1 0	P_2 1	1	P_3 1	0	1	0	P_4 0	1	1	0

P_2 covers all "xx1x" positions s.t. they make even parity
* $2^p \geq p+d+1$!

* Furtherly, create overall parity bit P_A for above 11 bits.

Transmit ⟶ Check:
- ① 每个Group ("xx1x") 之和为偶, 12位总和为偶 ⟹ NO ERROR!
- ② 有Group的和为奇, 但12位总和为偶 ⟹ 2, 4, 6 ... # of errors detected 不完成改正!
- ③ 有Group和为奇 (Ex: Group "1xxx" 和 "xx1x" combine), 12位总和为奇 ⟹ 1, 3, 5 ... # of errors occur. 完成假设只有 1 error, 改正.
  Ex: Error position ⟹ 1010 here. Flip this bit.

\+ 一定能检出错误.
\- 不一定改对.

进制表

Hex 16	Oct 8	Bin 2	Dec 10
0	00	0000	0
1	01	0001	1
2	02	0010	2
3	03	0011	3
4	04	0100	4
5	05	0101	5
6	06	0110	6
7	07	0111	7
8	10	1000	8
9	11	1001	9
A	12	1010	10
B	13	1011	11
C	14	1100	12
D	15	1101	13
E	16	1110	14
F	17	1111	15

小数表

i	Frac. 2^{-i}	Binary
0	1.0	1.0
1	0.5	0.1
2	0.25	0.01
3	0.125	0.001
4	0.0625	0.0001
5	0.03125	0.00001
6	0.015625	0.000001
7	0.0078125	0.0000001

Boolean 表

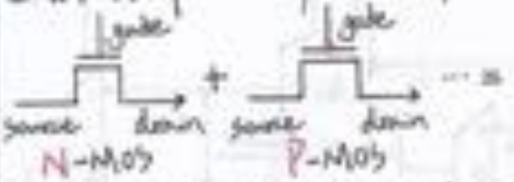
$X \cdot \overline{X} = 0$	$X + \overline{X} = 1$
$X \cdot 0 = 0$	$X + 0 = X$
$X \cdot 1 = X$	$X + 1 = 1$
$X \cdot X = X$	$X + X = X$
$X \cdot Y = Y \cdot X$	$X + Y = Y + X$
$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	$(X+Y)+Z = X+(Y+Z)$
$X(Y+Z) = X \cdot Y + X \cdot Z$	$X + Y \cdot Z = (X+Y) \cdot (X+Z)$
$X + XY = X$	$X \cdot (X+Y) = X$
✓ $X + \overline{X}Y = X + Y$	✓ $X \cdot (\overline{X} + Y) = XY$
✓ $\overline{X \cdot Y} = \overline{X} + \overline{Y}$	✓ $\overline{X+Y} = \overline{X} \cdot \overline{Y}$

次方表

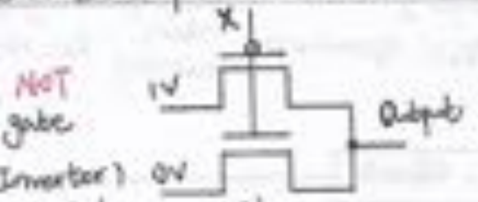
i	2^i Dec	
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1024	KiB
20	1024^2	MiB
30	1024^3	GiB
40	1024^4	TiB

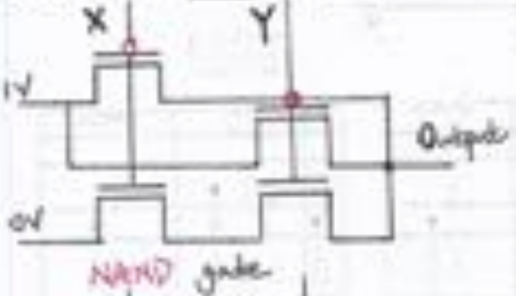
Synchronous: Coordinated by a center clock signal (CLK)
Digital: Represent values in binary
- Removes noises!

CMOS (Complementary MOS)-pairs!



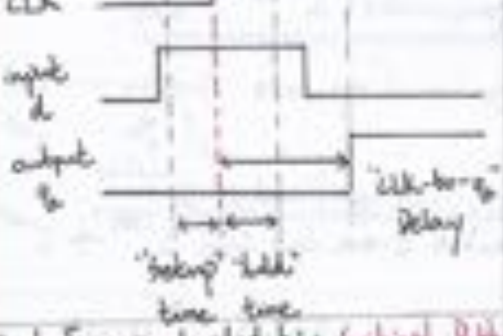
On: $V_{gate} > V_{thre}$ On: $V_{gate} < V_{thre}$
Use to pass 0 Use to pass 1
Never leave a wire undriven!
Never create path from V_{DD} → ⏚!





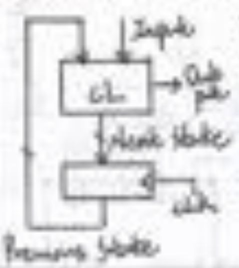
Logical function with N inputs: 2^{2^N}
SoP (Sum of Products): 取 Output 为 1
PoS: 取 Output 为 0, 以 "与" 再 "或" 的形式.

CL (Combinational logic):
Function of instant inputs
SL (Sequential logic): registers
Remembers historical information



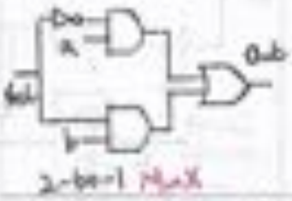
Clock Freq. is decided by: Critical Path between 2 registers

FSM (Finite State Machine)



P1	P0	In	N1	N0	Out
0	0	0	0	0	0
⋮			⋮		
x	x	x	x	x	x

Converts into Truth Table

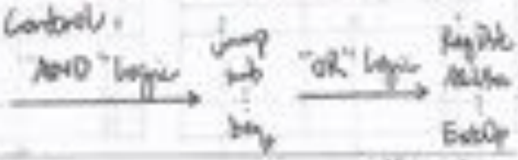


4-to-1 MUX just use three 2-to-1 MUX

Adder Logic   ALU (Arithmetic Logical Unit)
$Sum_i = XOR(a_i, b_i, c_i)$
$Carry\ into_{i+1} = a_ib_i + a_ic_i + b_ic_i$
$= MAJ(a_i, b_i, c_i)$
Conditional Inverter: XOR

Datapath:
- Can have stages idle
- 'Load' uses all 5 stages
- 'J' use the fewest stages

Control:



Pipelining requisite: Different resources.
+ Overall throughput
- Doesn't help latency of single task
(Actually slowing down since extra registers)
Speed up ⟶ # of stages
Actually <:
- Time to "fill" and "drain"
- Limited by the slowest stage
T_c: time between completion
$T_{c,pipelined} \geq \frac{T_{c,single\text{-}cycle}}{\#\ of\ stages}$
= Only when stages balanced.

1. Structural Hazard
Busy resources shared among stages
+ Can always be solved by adjusting or adding hardware.
- Split Inst. / Data memory.
- RegFile access halves to read + write in one CLK.

2. Data Hazard
Data flow backwards in time
+ May be solved by forwarding
- MUST stall instruction dependent and right after 'Load'
- "nop" / "bubble" / load delay slot
- Reorder code, put an unrelated Inst. right after 'Load'

3. Control Hazard
whether to jump / branch?
- May stall every Inst. after them
+ Add branch comparator early
+ Predict, "flush" pipeline when wrong
+ branch delay slot:
- Put a previous unrelated Inst.

Hyperthreading: Duplicate registers but use same CL circuit

Temporal Locality: in time
Spatial Locality: in space
Memory access with cache ($)
Process Address (32-bit)

Tag	Set Index	Offset

Size of Index = $\log_2$(# of sets)
Size of Offset = $\log_2$(# of bytes per block)
Valid bit: if 0, always miss
Total capacity = Associativity × # of sets × Bytes per block
Average Memory Access Time
AMAT = hit time + miss rate × miss penalty

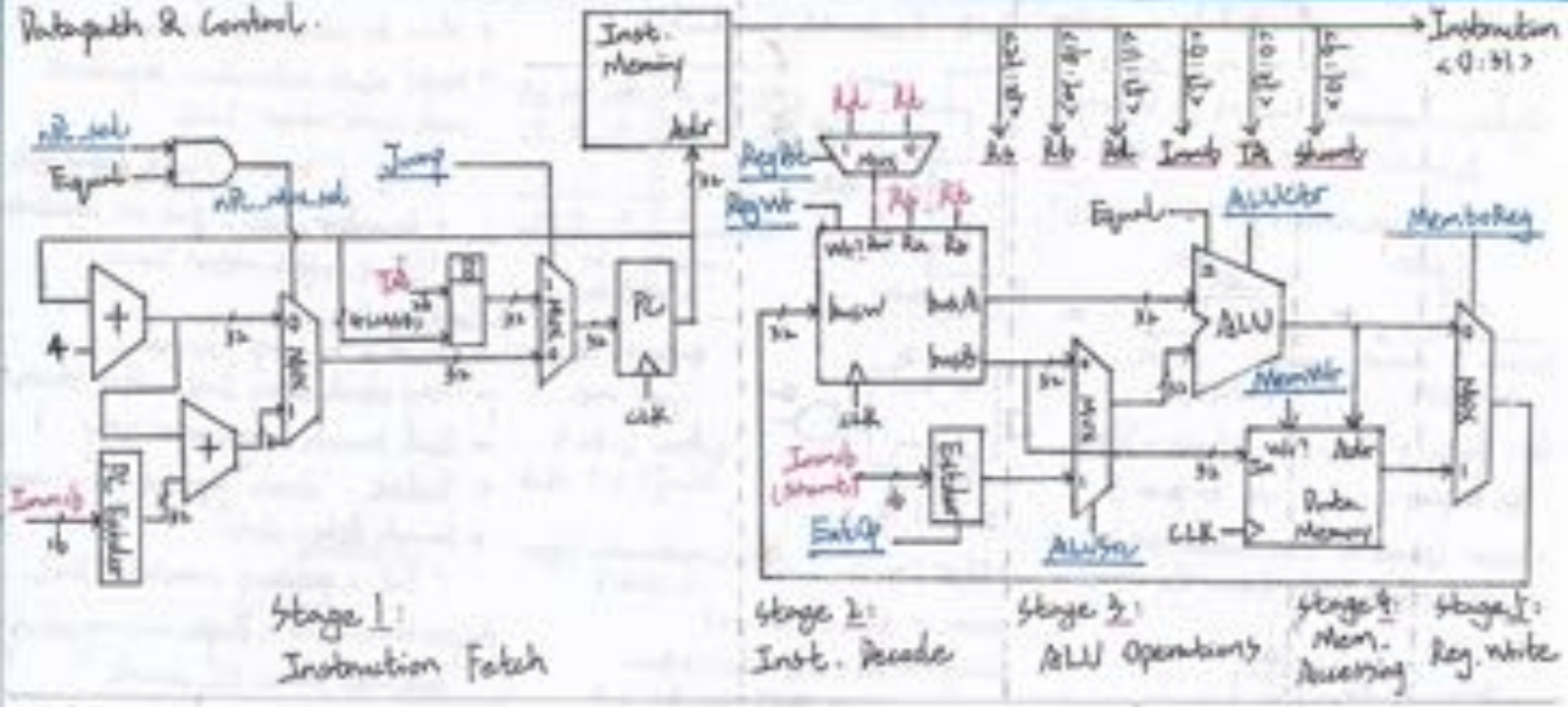
Write Through + Buffer	vs.	Write Back + Write Allocate
- Traffic busy		- Dirty Bit
+ Simple		- Complex
+ Redundancy ⇒ Reliable		+ Traffic liter

Direct Mapping vs. Associativity
- Ping-Pong effect   - More compare
+ # of comparators ↓   + Miss rate ↓

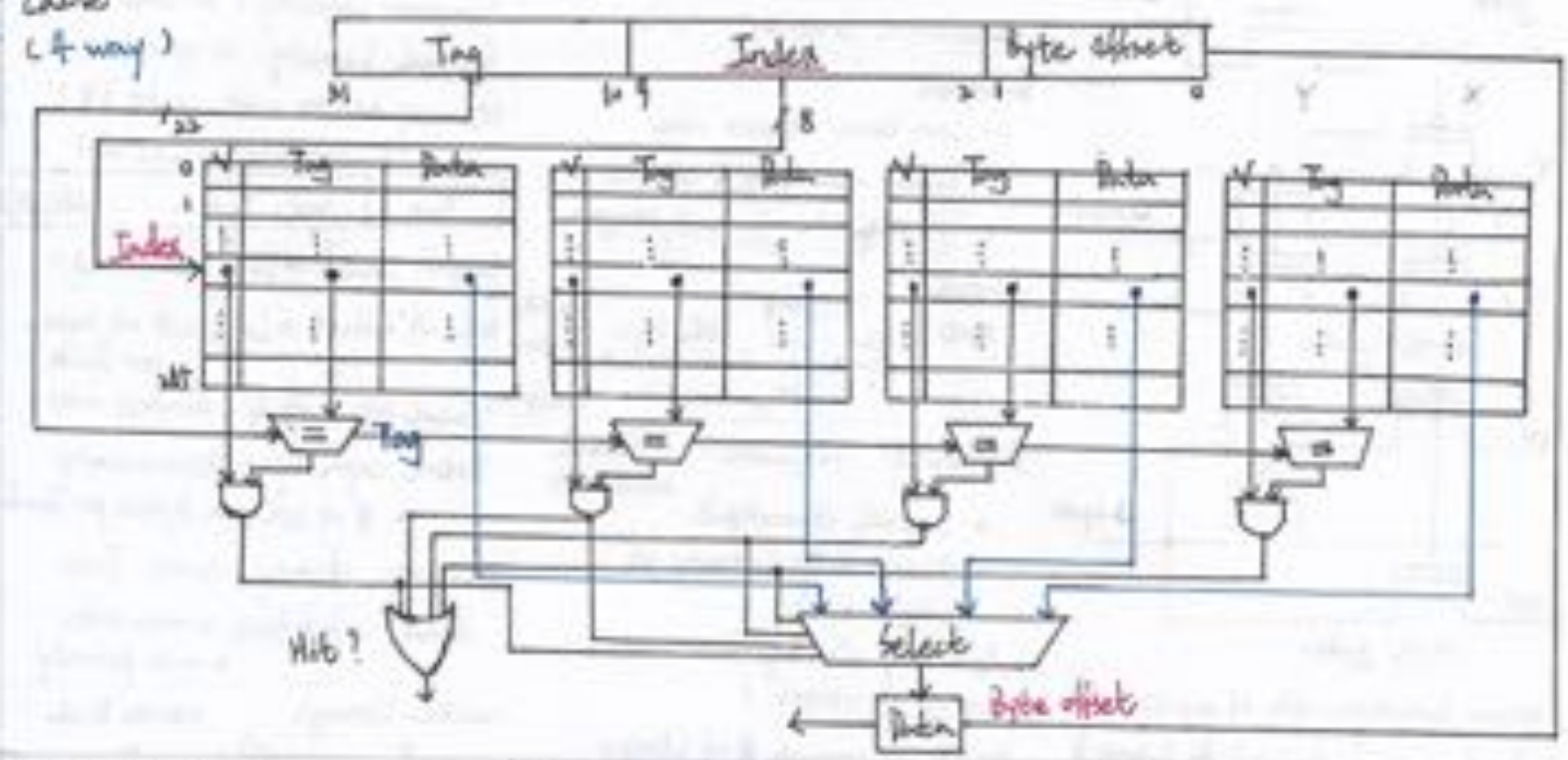
"3C" Misses
Compulsory: First access must fail.
Capacity: Cache full.
Conflict: Map to same cache block.

Figure Datapath / Cache Arbiter
:-)

Datapath & Control



Cache (4 way)



Multilevel Cache

Local miss rate $r_{\$2,local}$ = $\frac{\#\text{ of }\$L2\text{ misses}}{\#\text{ of }\$L1\text{ misses}}$

Global miss rate $r_{\$2,global}$ = $\frac{\#\text{ of }\$L2\text{ misses}}{\text{Total Accesses}}$

= $r_{\$1,local} \times r_{\$2,local}$

AMAT = Time for $1 hit + $r_{\$1,local}$ × (Time for $L2 hit + $r_{\$2,local}$ × $L2 Miss penalty)

Latency: Time for one task

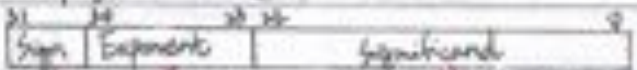
Bandwidth: Tasks per unit time (throughput)

$$Speedup_x = Performance_x = \frac{1}{Execution\ Time_x}$$

$$CPU\ Time = \frac{Inst.\ \#}{Program} \times \frac{clk\ cycles}{Instruction}\ (CPI) \times Time\ in\ a\ cycle$$

Workload: Set of programs running | Benchmark: Workload chosen for compare

IEEE 754 F.P.



Sign: { 0 - positive, 1 - negative }

$(-1)^S \times (1 + Significand) \times 2^{(Exponent - bias)}$

Expo	significand	means...
0	0	0
0	sth.	Denorm
1~254	anything	fl. pt #
255	0	+/- ∞
255	sth	"NaN"

significand represents: $0.e_{-1}e_{-2}\cdots e_{-23}$ (2^{-1}, 2^{-2}, 2^{-23})

$(-1)^S \times (0 + Significand) \times 2^{-126}$

makes 0 → a Gap from $2^{-126} \to 2^{-149}$

SISD vs. SIMD vs. MIMD

Intrinsic... Multicore...

Threads in one process shares memory space!

- Software Multithread ⇒ Time-sharing
- Hardware — (Hyperthreading) ⇒ Context-Switch Time saved...
  (Looks like 2 processors)
- Multithreading ⟶ Better Utilization
- \+ Multicore ⟶ Replicate Processors

OpenMP -tapemp

+ Shared memory. Spot
+ Compiler directives. easy to use
+ Serial code no need to be rewritten

- MUST be shared memory structure
- Compilers MUST support OpenMP

except loop indicator

pragma omp parallel, default is shared
OR private(A)
% declared inside parallel region, private

omp_set_num_threads(4); . set parallel threads
omp_get_num_threads(); . get # of threads
omp_get_thread_num(); . get thread ID

pragma omp for ← Do NOT break this for!
critical ; reduction (+/*... : var)

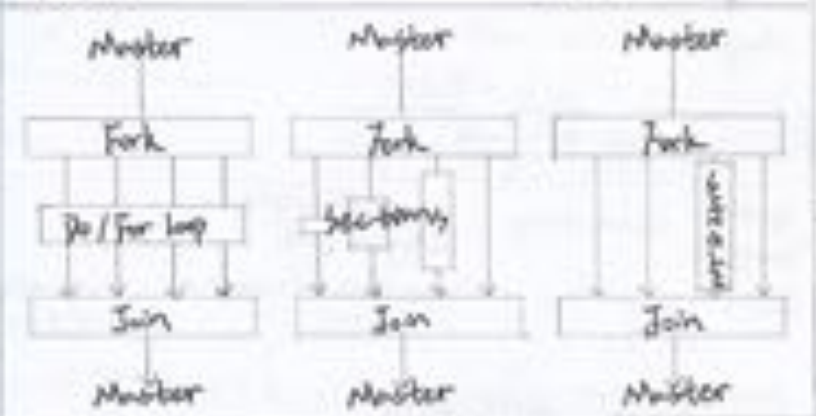
Atomic Hardware-supported Instructions

Test & Set

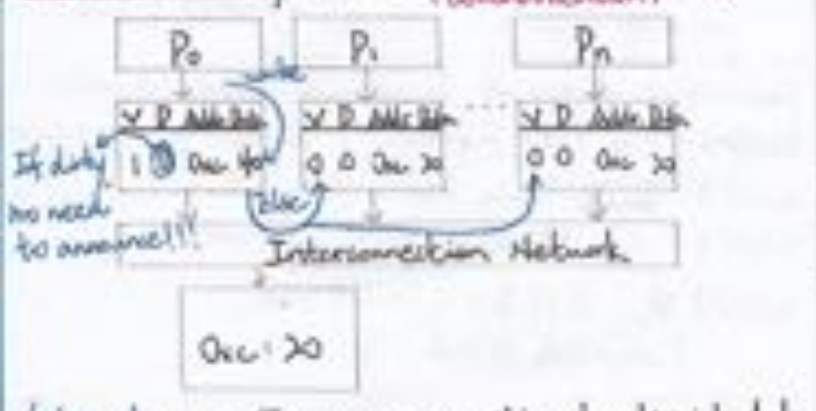
1. Test to see if is set?
2. If init. Set it.
3. Else. return that failed.

Example

ll $t1, 0($t0)
sc $t0, 0($t0)
changed? 0 : 1



Cache Coherency: 4th "C" Coherence (Communication) MMS



false sharing: Two processors writing to disjoint data which accidentally stay in one block.

Embedded Systems → X General Purposes

- Specialty-functioned
- Tightly constrained { Timing performance / Power consumption
- Reactive & Real-time
- HW-SW coexistance

$P_{dynamic} = \alpha \cdot C \cdot V_{dd}^2 \cdot f_{clk}$ ← Dynamic, Short-circuit, Leakage

CPU	⟷	FPGA	⟷	ASIC
Programmable		Configurable		Expensive/Tiny
Power ↑↑		Power ↑		Power ↓↓
Perf ↓↓		Perf ↑		Perf ↑↑

Routing Switches ⟷ Interconn. Functional Blocks LE < LUT Table, Technology Mapping

Warehouse Scale Computing (WSC)

+ Scale of economy
- High # of Failures

{ Blades (1U Server) / Rack / Array (Clusters)

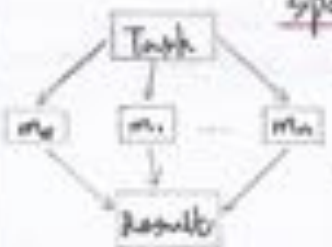
Power Usage Effectiveness

$PUE = \frac{\text{Total Building Power}}{\text{IT Equipment Power}} \geq 1.0$

Energy = Power × Time; Efficiency = Computation / Energy

Request level Parallelism (RLP) - Independent
Data level Parallelism (DLP) - Map & Reduce

Spark



- Map | Transform: RDD → RDD
- Reduce | Action: RDD → value

map, reduce, reduceByKey, flatMap, ...

Operating Systems (OS)

#1. I/O { Special I/O instructions / Memory-mapped I/O

Certain portion of address space corresponds to registers in I/O devices

Speed of CPU >>> I/O

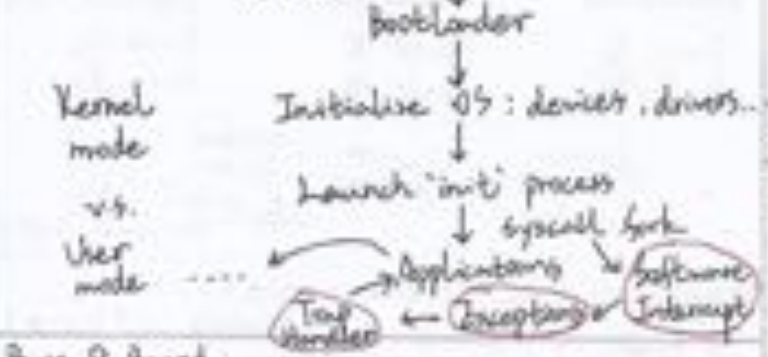
"Polling": Loop to read Control Register

Cost = % process time to poll =

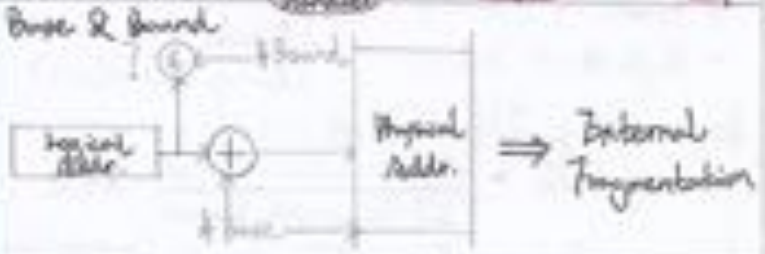
$$\frac{\#\text{ of polls}}{\text{sec}} \times \frac{\#\text{ of cycles}}{\text{poll}} \Big/ \frac{\#\text{ of cycles}}{\text{sec}}$$

"Interrupts":

#2 Bootstrap: Some start address stored in ROM (BIOS)

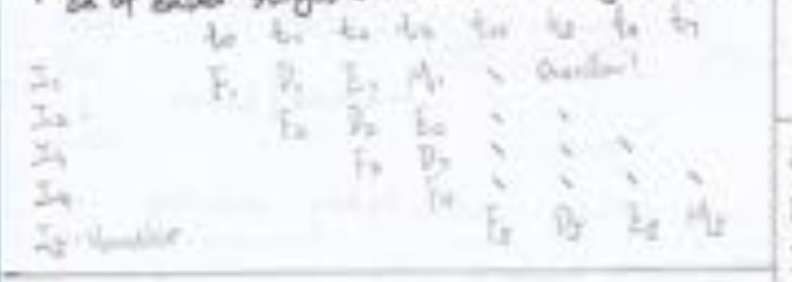


Base & Bound



Pipelined Precise Traps Handling

- Commit Point only at M stage
- In of earlier instructions override later instructions
- In of earlier stages override later stages for one



Virtual Memory

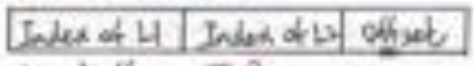
+ Adds disks into hierarchy
+ Simplify memory management for Apps
+ Protection { User ↔ Kernel; User1 ↔ User2

Page Tables reside in Main memory, per process.

- Linear: Set active proc's base Reg + #. as index

$$\text{Size} = \frac{\text{Size of User Space}}{\text{Size of a page}} \times \text{Size of an PTE}$$

- Hierarchical:



Translation Lookaside Buffer (TLB)

+ As a cache for page translations.
- On context switch, must flush away.

TLB Reach = Total size of VM space that can be mapped

= # of TLB entries × Size of a page

Page Fault: Found that a page not in memory now.

↳ A precise trap to bring in from disk

Virtual Machine: User — Kernel — VM mode

Direct Memory Address (DMA) v.s. programmed I/O

+ General Purposed CPU can leave data sending tasks to DMA device
+ CPU obtains spare time computing
+ Save energy

- Burst Mode: CPU out
- Cycle Stealing Mode
- Transparent Mode: Only when CPU not accessing

Magnetic Disks (HDD)

Access time = Seek + Rotation + Transfer + Ctrl overhead

Average: $\frac{\#\text{ of Tracks}}{3}$; $\frac{\text{Cycle}}{2}$

Spatial / Temporal Redundancy - Replicated Copies
- Retry in time

Mean time to Fail (MTTF) + Mean time to repair (MTTR) = ... between (MTBF)

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF}+\text{MTTR}} \times 100\% : 3\text{"9"s} = 99.999\%$$

$$\text{Annualized Failure Rate (AFR)} = \frac{\#\text{ of Failed}}{\text{Total \#}} = \#\text{ of Hours in a year (8760)} / \text{MTTF in hours}$$

Redundant Arrays of Inexpensive Disks (RAID)

RAID 1: Disk Array "mirroring" / "shadowing", expensive

RAID 3: Single Parity Disk.

RAID 4: High I/O Rate Parity.

RAID 5: Parity Disk

Ecc. Hamming Codes on cheatsheet 1 ;)

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data

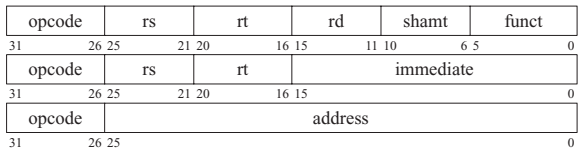
①

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

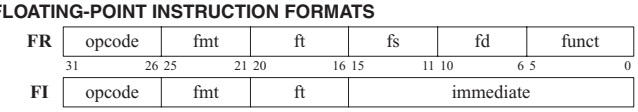
②

NAME, MNEMONIC		FORMAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0

(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127, Double Precision Bias = 1023.

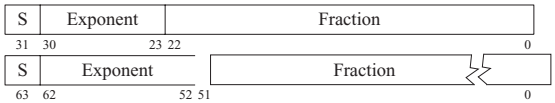
④

IEEE 754 Symbols

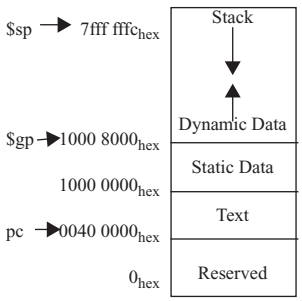
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

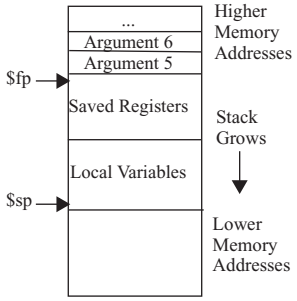
IEEE Single Precision and Double Precision Formats:



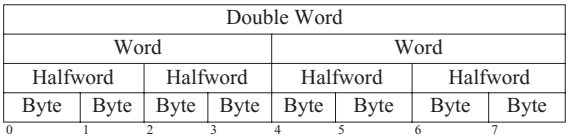
MEMORY ALLOCATION



STACK FRAME

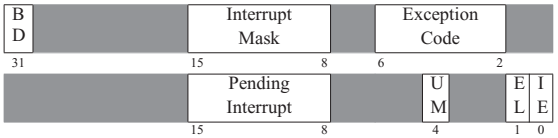


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add, addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi, addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch if Equal	if(R[rs1]==R[rs2]) PC=PC+{imm,1'b0}	
bge	SB	Branch if Greater than or Equal	if(R[rs1]>=R[rs2]) PC=PC+{imm,1'b0}	
bgeu	SB	Branch if ≥ Unsigned	if(R[rs1]>=R[rs2]) PC=PC+{imm,1'b0}	2)
blt	SB	Branch if Less Than	if(R[rs1]<R[rs2]) PC=PC+{imm,1'b0}	
bltu	SB	Branch if Less Than Unsigned	if(R[rs1]<R[rs2]) PC=PC+{imm,1'b0}	2)
bne	SB	Branch if Not Equal	if(R[rs1]!=R[rs2]) PC=PC+{imm,1'b0}	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1'b0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[63],M[R[rs1]+imm][7:0]}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm][7:0]}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm][63:0]	
lh	I	Load Halfword	R[rd] = {48'bM[15],M[R[rs1]+imm][15:0]}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm][15:0]}	
lui	U	Load Upper Immediate	R[rd] = {32'bimm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[31],M[R[rs1]+imm][31:0]}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm][31:0]}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm][7:0] = R[rs2][7:0]	
sd	S	Store Doubleword	M[R[rs1]+imm][63:0] = R[rs2][63:0]	
sh	S	Store Halfword	M[R[rs1]+imm][15:0] = R[rs2][15:0]	
sll, sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli, slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra, sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai, sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl, srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli, srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub, subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm][31:0] = R[rs2][31:0]	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit registers
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., -inf, -0, +0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]<0) ? -F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = -F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = -R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul, mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div, divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem, remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu, remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64A Atomic Extension

MNEMONIC	FMT	NAME	DESCRIPTION	NOTE
amoadd.w, amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w, amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w, amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w, amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w, amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w, amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w, amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w, amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w, amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w, lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w, sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 … 27	26 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

REGISTER NAME, USE, CALLING CONVENTION ④

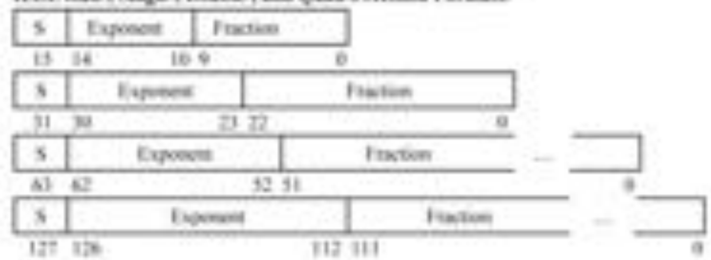
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Caller
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Caller
x9	s1	Saved register	Caller
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Caller
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Caller
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Caller
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

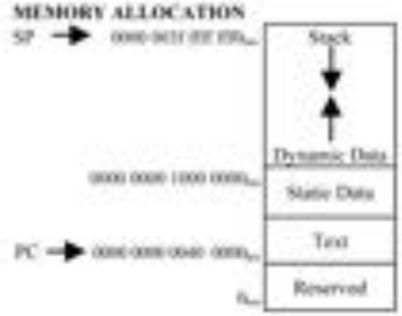
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

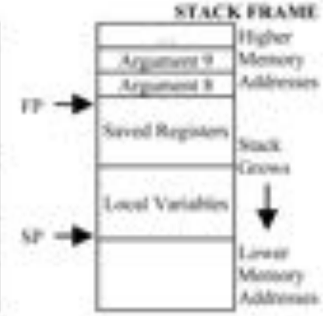
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y