61C Crib Sheet – DIVIT RAWAL

Int Representations

		#Vals	#0	Notes
Biased	$[B, 2^n-1+B]$	2^n	1	
2's Compl.	$[-2^{n-1}, 2^{n-1}-1]$	2^n	1	simplifies arithmetic
Sign Mag.	$[-2^{n-1}+1, 2^{n-1}-1]$	2^n-1	2	more complex
1's Compl.	$[-2^{n-1}+1, 2^{n-1}-1]$	2^n-1	2	more complex
Unsigned	$[0, 2^n-1]$	2^n	1	

2's compl: $-x = x$ flip all bits, add 1
↳ MSB 0 ⇒ +tive

Memory: Stack: LIFO, auto managed, grows ↓, contig, SP @ bottom, sub to move up
funcs get → stack frames. Heap: dynamic, not contig, man alloc
cannot return ptr to stack
↳ Errs: mem. leak, dangl ptrs, double free, realloc data but leaving ptr.
(i) addrs. in bytes, (ii) word = 4 byts, (iii) 8 bits < data < 32 bits

Endianness

0xABCDEFAD (MSB … LSB)

	BIG
a	AB
a+1	CD
a+2	EF
a+3	AD

load from a → a+3

0xABCDEFAD (MSB … LSB)

	LITTLE
a	AD
a+1	EF
a+2	CD
a+3	AB

load from a+3 → a

RISC-V: 1 instr. = 32 bits

- R type – 3 reg, 0 imms
- I type – 2 reg, 1 imm
  - ↳ imm has 12 bits
  - ↳ [-2048, 2047]
- I* type – 2 reg, 1 imm
  - ↳ 5 bit imm = max 31 bitshift
- S type – 2 reg, 1 imm
  - ↳ 12 bit imm [-2048, 2047]
- U type – 2 reg, 1 imm
  - ↳ 20 bit imm
  - ↳ discards bottom 12 bits: lui t0 0x12345 = 0x12345000
- B type – 2 reg, 1 imm
  - ↳ 13 bit imm [-4096, 4094]
  - ↳ 2^{10} instrs up/down max
- J type – 1 reg, 1 imm
  - ↳ 21 bit imm
  - ↳ 2^{18} instr up/down

Handling Large Imm
- I-type: store imm → temp reg
- S-type: add offset to imm (0 offset load)
- B-type: <1024: beq t0, t1, label
  - >1024: invert → bne t0, t1, next; do j → j label
- J-type: <1024: j label next
  - >1024: auip c → auip t0 0x12345; jalr imm t0 → jal ra t0, 0x678; offset remainder

GDB
- r ← run
- c ← continue
- finish ← guess brkn
- b fn ← break
- q ← quit
- break fn:ln ← break @ filename line
- n ← next, step
- p x ← print x
- s ← step, step in
- break loc if cond

$(-1)^S \times 1.M \times 2^\wedge(E\text{-bias})$ Floating Point

Category	S	E val	M val	total #	Details
Positives	0	any	any	$2^{E+M}-1$	excl. 0, inc NAN, den, ∞
Negatives	1	any	any	$2^{E+M}-1$	excl. 0, inc NAN, den, ∞
SPECIAL					
NANs	0/1	all 1s, val: 2^E-1	not all 0, $[1, 2^M-1]$	$2(2^M-1)$	-1 to excl all 0 in M, ×2 for +/- NAN
0s	0/1	all 0's, val: 0	all 0s, val: 0	2	×2: +/- 0s
∞	0/1	all 1s, val: 2^E-1	all 0s, val: 0	2	×2: +/- 0s
denorms.	0/1	all 0s, val: 0	not all 0, $[-1, 2^M-]$	$2(2^M-1)$	-1 to excl. all 0 in M, ×2 for +/- denoms

Total possible vals: $2^{E+M+1} - 2(2^M-1)$

Range for n bits: $[2^{B+1-M}, (2-2^{-M}) \times 2^{-B}]$
- exponent = all 0s = bias; mantissa = all 0 except for rightmost 2
- ↑ mantissa all 1s
- ↑ exp, all 1 except for 2^0

Bias: $-(2^{n-1}-1)$ smallest unrep. int = $2^{M+1}+1$

Normalized $(-1)^S \times 2^{E-B} \times 1.\text{significand}_2$

Denormalized $(-1)^S \times 2^{E-B+1} \times 0.\text{significand}_2$

Step sz: min diff b/w 2 adj reprable fp #s around a val. $= 2^{E-B-M}$ for M bits mantissa, E exp. field, B bias

Bitwise Ops
- & – masking
- | – set specific bits to 1
- ^ – toggle bits, compare diffs
- ~ – invert bits
- $x \ll n$ – LSB → MSB, $x \cdot 2^n$
- $x \ll n = x \cdot 2^n$
- $x \gg n = x / 2^n$ (uns)

LOGICAL SHIFT ⇒ NO SGN EXT
↳ PAD W/ 0
ARITHMETIC R PADS SGN

Bitwise Useful

Swap upper & lower bits of byte (4 ↔ 4)

$x = ((x \& 0xF0) \gg 4) | ((x \& 0x0F) \ll 4)$

Slide over bits

```
loop: # 2bit sliding window
  andi t1, a0, 3 # 2 LSBs
  # check smth
  srli a0, a0, 1
  bnez a0, loop
```

dirty indicates not in mem only in cache

FOR Recursive RISC-V, always save/load ra !! ←

```
addi sp sp -4
sw ra 0(sp)
j
lw ra 0(sp)
addi sp sp 4
```

WB is complete
C ~~…~~ cycle of last stage

VPN indexes Page Table

CALLING CONVENTION

- ra: when in fn, holds addr of line imm after
- a0-a7: hold args, a0, a1 for return vals, set before call jal
- sp: bottom of stack, anything below safe to use. DO NOT use neg offsets, dec sp then use pos off.
- t0-t6: may change after fn call
- s0-s11: must stay same after fn call
- Caller-saved: ra, a0-a7, t0-t6, may change after fn call
- Callee-saved: s0-s11, sp, must not change after fn, save in prol, epil.

Fn callin
1. setup args (a0-a7)
2. Control to fn jal ra fn
3. Prologue
4. do task
5. Epilogue
6. Return control jr ra

strlen goes till \0

C

Compilation
input = c(foo.c)
↓
Preproc.: macros
↓
foo.i
↓
compiler + assembler
↓
foo.o lib.o
↓
Linker
↓
a.out.

ALWAYS ASS LIL ENDIAN

Data Sizes:

Type	Bytes
char	1
Short	2
long long	8
int	4
float	4
double	8
ptr.	4

Ptr to func.
<rettype>(*name)(argtypes, ...) = & funcname

Struct
each member has own space mem. . vs →
Pad to size of largest dtype, think stacking
typedef struct { dtypes, names } name;

Unions
membs share mem space, only one memb can have val @ a time
size = largest elem

Pointers (4 bytes)
dtype * name = & item
*pname ← deref: val
&addr
((as-type) addr)
type ** p2p = & ptr.
** p2p = val
ptr inc. by dtype size in bytes
pname += 1 ; void* general

Arithmetic
ptr+n ≡ +n * sizeof(type) to addr
ptr-n ≡ -n * sizeof(type) to addr

Arrays
<type> name [#];
type name [#] = {vals};
type (*name)[#];
pass into funcs: pass ptr to arr start.

Strings
Static immutable char *a = "string";
↳ w[0] = 'p' fails
↳ w = "p" works
heap mutable char *a = malloc(sizeof(char) * strlen("...")+1)
stack mutable char a[] = "str"
strlen does not count \0
sizeof(str) includes \0
strcpy(..) copies w/ \0

MemAlloc
malloc() → ptr; garb.
calloc() → ptr; 0 init
realloc() → ptr; garb.

RISC-V Memorote
- inc addr in bytes
- 1 instr = 4 bytes
- labels not stored, addr of fn = addr of first instr
- initialize t b4 usage

Mem Storage
Stack: local vars in funcs.; ptrs to str consts; func params; stack arrs
earlier dec → higher addr
later dec → higher addr
Heap: mallocs / reallocs / callocs
Static: immutable / read-only
r/w: global vars, arr cnst.
Code: funcs outside main; preproc directives, defines; consts, main

Caching (more!)
locality: temporal: MRU
Spatial: contig blocks

Tradeoffs:
Bigger block: Spatial locality ✓ larger miss penalty
more blocks: temporal locality ✓ higher miss rate

Max Sz: $2^{off} \times 2^{idx}$ blocks
Min Sz: 2^{idx} blocks

Masks
1. NOT: XOR 11111111_2
2. LSB: andi 0x000000FF
MSB: andi 0xFF000000

	logical	arithmetic
instr	srli, sll, slri, slr	sra srai (def=logical)
replace	insert 0	sign extend

AUIPC
32 bits
1. shift imm 3 words left
2. add imm to PC
3. saves in dest reg = rel. addressing

LUI
Load imm bigger than 20 bits
1. check if 11th bit is 1
2. split imm → 5 words → 3 words
3. add 1 to the 5 word grp.
↳ inc. hex letter by 1
4. LUI 5 words → upper 5 words
5. add 3 words

CALL
Compiler: .c → .s
Assembler: .s → .o
↳ .text = code
.data = init vars
symbol table
Linker: .o + libs → .out
links shit
Loader: exec code in mem
sp
pc call-start
linker no output
pseudo instr.

RISC-V to Binary
R: op rd rs1 rs2
→ funct7 rs2 rs1 funct3 rd opcode
[7] [5] [5] [3] [5] [7]

I: op rd rs1 imm
→ imm rs1 f3 rd opcode
[12] [5] [3] [5] [7]

S: op rs2 imm(rs1)
→ imm rs2 rs1 f3 imm[4:0] opcode
11:5 [7] [5] [5] [3] [5] [7]

B: imm[12|10:5] rs2 rs1 f3 imm[4:1|11] opcode
[7] [5] [5] [3] [5] [7]

U: imm[31:12] rd opcode
[20] [5] [7]

J: imm[20|10:1|11|19:12] rd opcode
[20] [5] [7]
↳ dist b/w curr & targ. instr

Calling Convention
Args in a0-a7 & ret in a0, a1
t0-t6 temp, s1-s11 callee saved

Caching
Blocks - chunk of mem moved tg.
sets -
associativity - # blocks/set
Addr = [TAG][INDEX][OFFSET]
FA Cache: any mem block to any cache block
1 set total
must search all cache blocks
miss rate usually low
Eviction Policies: LRU, FIFO, random
Write Policy: Through: write to cache & mem
Back: only to cache, update mem when evicted
↳ need dirty bit
Offset: byte w/in block
Index: selects set
Tag: identifies block

RISC-V Coding
① Prep stack: save regs → call fn → restore regs + del stack
dec sp; save ra, s0, save func args (a[0,...]) to s regs
② OFFSET MUST // 4 = 0
Load: load word lw x10, 12(x15)
load byte lb x10, 12(x15)
↳ load into lowest byte of x10 & sign ext.
Store: store word: sw x10, 40(x15)
store byte: sb x10, 40(x15)
LA: load addr of var into reg.
③ Increment: get start ptr, load byte @ ptr (lb t1 0(a0)); inc by sz (addi a0 a0 1)
④ Branching: [str] beq t2 x0 End
⑤ Jumping:

	Don't	Come Back
jump to lab	j label (jal x0 lab)	jal ra label
j to addr in reg	jr ra (jalr x0 ra)	jalr ra s0

ret ≡ jalr x0, ra, 0
jal ra label: PC jumps to label ↳ saves ra

⑥ Call Fns.
1. mv s0 a0 (save args)
2. mv a0 t0 (args to arg. regs)
3. addi sp sp +-# (stack space)
4. jal ra fn lab (call fn)
5. mv t0 a0 (store return val)
6. mv a0 s0 (restore regs)
7. return control to call pt.

⑦ Recursion
1. mv s0 ra (save ra to stack)
2. lb a0 0(t0) (set arg regs)
3. jal ra fn_y (call fn_y, save x as ret addr, x = ra)
4. mv ra s0 (restore ra)
5. jr ra (fn x ret)

⑧ Closing
1. reload all saved regs from stack
lw s0 4(sp)
2. addi sp sp 4 (reset sp)
3. jr ra (return)

offset = $\log_2$(# blocks)
Direct Mapped Caches: 1 block / set, index = $\log_2$(#sets)
↳ High miss rate due to collisions
N-Way Set Ass. Caches: N blocks / set
Miss Classif.
Compulsory: first time access (cold miss)
Capacity: cache too small to hold everything
Conflict: 2 blocks map to same set

AMAT: HitTime + MissRate * MissPenalty

RISC-V Datapath

Instruction Fetch (IF)
↳ PC sel ≠ *
  ↳ 0 ≡ PC+4
  ↳ 1 ≡ PC ≡ ALU output

Instruction Decode (ID)
↳ ImmSel (can be *)

Execute (EX)
↳ ASel (can be *)
↳ BSel (can be *)
↳ ALUSel (rarely *)
↳ Branching (can be *)
  ↳ BrUn
    ↳ 0 (unsigned comp)
    ↳ 1 (signed --)
  ↳ BrLt
    ↳ 0 (A ≥ B)
    ↳ 1 (A < B)
  ↳ BrEq
    ↳ 0 (A ≠ B)
    ↳ 1 (A = B)

Mem Access (MEM)
↳ MemRW (should never be *)
  ↳ 0 do not write
  ↳ 1 (store instr)

Writeback (WB)
↳ WBSel (can be *)
  ↳ 0 (mem data to reg) (loads)
  ↳ 1 (ALU output to reg)
  ↳ 2 (sends PC+4 to reg)
    ↳ useful for saving addr of next instr for returning from a jump
  ↳ 3 (useless)
↳ RegWEn (never *)
  ↳ 0 (no wb to regfile allowed even if reg write val is set)
  ↳ 1 (writes to regfile)

Pipelining

Iron Law
Throughput = #instrs/time
Latency = time/instr
Pipelining only increases throughput, not latency
↳ latency may increase

$\frac{Time}{Prog} = \frac{Instrs}{Prog} * \frac{Cycles}{Instr} * \frac{Time}{Cycle}$

Hazards

Structural: HW does not support access across multiple instrs. in same cycle
Data: Instrs need to wait for prev to finish R/W
↳ fix w/ stalling or forwarding
Control: flow of exec. depends on prev. instr.

Freq = $(\text{clk period})^{-1}$
clk per ≥ clk-q delay + longest combinatorial delay + setup time
Hold time: how long the input must be stable after rising edge of clk
Setup time: how long input must be stable before rising edge of clk

Parallelism

Amdahl's Law
Speedup $= \frac{1}{(1-F)+(\frac{F}{S})}$
F ≡ % of code sped up
S ≡ speedup factor
denom is new running len of code

Loop Unrolling: multiple iterations in a single iter of loop
↳ need tail case for mod
↳ reduces control hazards

Fn Inlining, Var Caching

Virtual Memory

Virtual Addr - seen by os + CPU

VPN (Virt. Page #)	Pg offset

Physical Addr

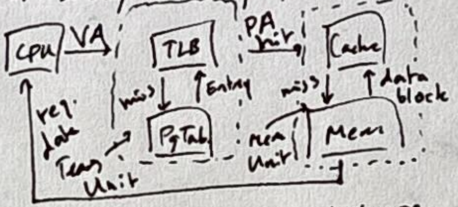
PPN (Phys. Page #)	Pg. offset

Virtual Page: consec sect of mem in virtual addr space
VPN: pg's idx in Virt Mem
Physical same but virt → phys
VA offset = PA Offset
#Pg. Offset Bits = lg(sizeof page)
↳ same for virt & phys.
#VPN Bits = lg(#pg. table entries) = lg(#virtual pgs)
#Bits in VA = lg(sizeof virt. mem. space) = #VPN Bits + #Pg Offset Bits
#PPN Bits = lg(#phys. pages)
#Bits in PA = lg(sizeof phys. mem. sp.) = #PPN Bits + #Pg Offset Bits
#Pg. Table Entries = #Virt Pgs. = $2^{\#VPN\ Bits}$
sizeof(virt. mem) = #virt pgs. * pg sz.
sizeof(phys. mem) = #phys. pgs. * pg. sz.

Page Table stored in Mem
↳ VPN is idx
  ↳ Entry
    ↳ PPN, metadata
  ↳ Valid Bit
    ↳ if 1 ⇒ in DRAM
    ↳ else in disk
      ↳ pg. fault

Translation Lookaside Buffer (TLB)

2 mem accesses needed for 1 data addr access
↳ slow
TLB is cache for VPN-PPN mappings
↳ usually small & fully assoc.



If status bit is not valid → pg. fault
↳ Alloc. free pg.

RAID 0
↳ no redundancy
↳ speed, failure etc
RAID 1
↳ duplicate data
↳ reliable, less capacity fast reads

check parity bits
0 = pass
1 = fail
p4p2p1 is idx of bad bit

Atomic Instrs: check val & write to mem @ same time
↳ useful for locks

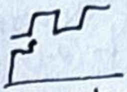
Misc.

$t_{clk\text{-}to\text{-}q} + t_{shortest\ path} \ge t_{hold}$

$t_{clk\text{-}to\text{-}q} + t_{longest} + t_{setup} \le t_{clk\text{-}per}$
↳ critical path b/w regs

State elems store vals for indeterminate amt of time

Flip-Flops



D data
Q output
Sample input if clk on rising edge
↳ ignored if not rising

FSMs

Label transitions I/O
↳ input → dest st.
↳ output → out caused by in
2 transitions from any state ⇔ 2 possible In

Facts

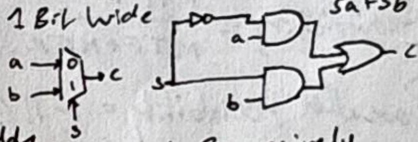
inc. clk per solves setup time violations
comb log ↑ → ↑ delay → sys may fail
↓ max log. delay → ↑ op. clk. freq.

Pipelining

add regs → more out/sec → ↑ clk freq.

MUX 2^n inputs, 2^{2^n+n} rows in TT

1 Bit Wide



Construct Recursively

Hamming w/ m Data Bits needs r parity bits

$2^r \ge m + r + 1$

Parity bit = ED, Hamming ~~detects~~ corrects 1B errs, detects 2B errs

Powers of 2 are parity bits

Parity bits cover all pos w/ 1 in binary rep of that bit index; e.g.

p1 001 ← covered by p1
p2 010 ← covered by p2
d1 011 ← covered by p1, p2
p4 100 ← covered by p4

XOR of all covered bits is 1 (odd parity)
XOR of all covered bits is 0 (even parity)

SIMD - most speedup comes from loading contig mem together
MIMD - threads share heap, indep reg, stack, PC
↳ #pragma omp parallel
↳ fork & join
Prevent data races: (a) don't change shared data
(b) critical secs - only 1 thread can do at a time (serialized)
#pragma omp critical

RAID 4

RAID 0 but 1 disk w/ parity for each stripe
↳ XOR (N-1) disks usable
↳ survive 1 disk failure

RAID 5

↳ RAID 4 but parity spread across disks
↳ (N-1) usable, reads parallel, writes better than R4, (parity spread out)

RAID 6

R5 but w/ 2 diff parity blocks / stripe
↳ (N-2) usable

Measures

MTTF: avg time sys operates before first failure

MTTR: avg. time to repair failure

MTBF: mean time b/w failures
↳ MTTF + MTTR

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

Parallel Reliability = $1 - \prod (1 - R_i)$

Series Reliability = multiply

Open MP

#pragma omp parallel

#pragma omp parallel for
↳ split its among threads

#pragma omp parallel for reduction (+: sum)
↳ give each thread its own copy. sum @ end
↳ prevent race conds.

#pragma omp barrier

pragma omp atomic

#pragma omp critical