Number Representation

TYPE	RANGE	LOWEST	HIGHEST	# of representable numbers
unsigned	$[0, 2^{n-1}]$	00...00	11...11	2^n
sign & magnitude	$[-(2^{n-1}-1), 2^{n-1}-1]$	11...11	01...11	$2^n - 1$ ← 2 reps of 0
2's complement	$[-2^{n-1}, 2^{n-1}-1]$	10...00	01...11	2^n
Biased	bias, $2^n - 1 + bias$	00..00	11...11	2^n

TWO'S COMPLEMENT: first bit = 1 → negative

To find magnitude of negative #:
1. invert bits
2. add 1
3. Negate unsigned result

shift left logical in 2's comp: multiply by 2
shift right in 2's comp: floor divide by 2

Bias: $x + b = n$ (b = bias, n = unsigned binary, x = # you want to represent)

```
comp_strings(**p, **q):
char *str1 = [*((char**)p)]
char *str2 = [*((char**)q)]
```

C (pass by value)

types: char: 8 bits = 1 byte int8_t = 8 bit signed
int: 32 bits = 4 bytes

False values: '\0', 0, NULL

structs: use (*struct).field
struct → field

pointers: use & to get address
use * to get value (reference)

pointer arithmetic: increment n by (ex: int = +4) sizeof(pointer_type)

arrays: int arr[] = {1,2,3};
char* hello = "hello";
a[i] → *(a+i)
char* hello = malloc(sizeof(char)* 6);

strlen does NOT include NULL; acct for null terminator

* Strings end w/ null terminator '\0'

Memory allocation:
malloc(size_t size): returns void* pointer, initializes w/ garbage (check if return NULL)
free(void* ptr): must free any malloc'd pointer once
realloc(void* ptr, size_t size): reallocate memory
calloc(num_items, size): allocates & initializes to zero

void* memcpy(void* dest, const void* src, size_t n)
void* memmove(void* dest, const void* src, size_t n)

Endianness:

BIG ENDIAN — 0A0B0C0D

a	0A
a+1	0B
a+2	0C
a+3	0D

MSB in LOWER address

RISC V 0D0C0B0A
LITTLE ENDIAN — 0A0B0C0D

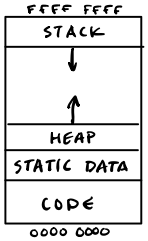
0D	a
0C	a+1
0B	a+2
0A	a+3

MSB IN HIGHER address

logical left shift: shift left add zeros = multiply pow 2
logical right shift: shift right zero extend = divide unsigned by power of 2
arithmetic right shift: shift right sign extend ≠ divide signed by pow of 2

slli x12 x10 0x10 shifts LEFT by 16 bits 0x00003AFF → 0x3AFF0000
srli x12 x12 0x08 shifts RIGHT by 8 bits 0x3AFF0000 → 0x003AFF00
srai a0 a0 0x02 shifts RIGHT by 4 bits (sign extend) 0xF0000000 → 0xFF000000

Memory Management



grows down (LIFO), function/local vars, erase when fxn return

int* foo = malloc(sizeof(int)) #DEFINE x
foo = stack *foo = heap x in code/text

dynamically allocated (malloc, calloc, realloc), bottom up, persists over calls

global vars outside of fxns, immutable, string literal, no grow/shrink

loaded @ start, executable read-only code, functions, machine instructions

Floating Point

32 bit Single precision
Bias = −127

sign	exponent	significand/mantissa
1 bit	8 bits	23 bits

Normalized: Value = $(-1)^{sign} * 2^{Exp+bias} * 1.significand_2$

Denormalized: Value = $(-1)^{sign} * 2^{Exp+bias+1} * 1.significand_2$

EXPONENT	SIGNIFICAND	MEANING
0	Anything	DENORM
1-254	Anything	NORMAL
255	0	Infinity
255	Nonzero	NaN

HEX → VALUE
0xFD0973E1
= 0b 1111 1101 0000 1001 0111 0011 1110 0001
sign = 1
exponent: 250 + (−127) = 123
significand: 1.00010010111001111100001
Value: $(-1)^1 * 2^{123} * 1.00010010111001111100001$

VALUE → Binary
−47.34375
① 47 = 101111
② 0.34375: 0.25 + 0.0625 + 0.03125 = $2^{-3} + 2^{-4} + 2^{-5}$ = 0.01011
③ 101111.01011 = 1.01111101011×2^5
④ x − 127 = 5
x = 5 + 127 = 10000100
1 10000100 01111101011 pad w/ zeros

- 25 bits mantissa → ints up to 2^{26} → CANNOT represent $2^{26}+1$
- floating point closest to 0 = denorms
- $2^{25}-1$ pos NaN, $2^{25}-1$ neg NaN
- most negative denormalized: −1 sign, exponent = 0, all 1s for significand

RISC V

callee: s0-s11, sp
caller: a0-a7, t0-t6, ra
save & restore

PC = program counter → where we are in running code
Jump instructions MODIFY PC

funct7, funct3, opcode = specify operation
R-type = arithmetic (2 source registers)
I-type = immediate arithmetic (1 source, 1 immediate)
I*-type = immediate shift (slli, srli, srai)
S-type = store
B-type = branches
U-type = upper immediate
J-type = jump (jal, j)

add more RISC commands

lb: load byte will go to LSB & MSB smears to fill
* lb x12 1(x5) = 0x93 → 0b1001... → 0xFFFFFF93

lbu: unsigned smears w/ 0 → 0x00000093

andi with 000000FF to isolate LSB

lui: writes 8 bits to LSB and fills other 24 bits w/ 0

auipc: build 32 bit with a lui
** add to this

	jump there, DON'T COME BACK	jump there, COME BACK
jump to label	j label (pseudo for jal x0 label)	jal ra label (ra = destination register w/ address to return to)
jump to address	jr ra (pseudo for jalr x0 ra 0)	jalr ra s1 0 (jump to 0 + address in s1)

CALL

gcc = c compiler

src: foo.c
↓ c processor
intermediate files
↓ COMPILER
assembly files: foo.s
↓ ASSEMBLER
Object files: foo.o
↓ LINKER
executable files foo.out
↓ LOADER (OS)
↓ Memory

C — C → RISC V
outputs labels, pseudoinstructions, relative addressing
error checking, reorders code

A — RISC V → object files
replaces pseudoinstructions w/ assembly
relocation table (external labels)
symbol table (labels, static data, relative addresses)
arithmetic, logical shifts

L — object files → single binary executable
put tgt data & text from .o files → put on end of text
resolve refs w/ reloc table → fill in addresses

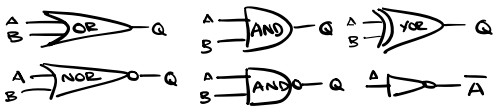
L — load program into memory for execution
initialize registers
runs a.out
part of operating system

SDS

reg → CL → REG

	AND	OR
commutative	AB = BA	A+B = B+A
associative	A(BC) = (AB)C	A+(B+C) = (A+B)+C
identity	1A = A	0+A = A
null	0A = 0	1+A = 1
absorption	A(A+B) = A	A+AB = A
distributive	(A+B)(A+C) = A+BC	A(B+C) = AB+AC
idempotent	A(A) = A	A+A = A
inverse	A($\overline{A}$) = 0	A+$\overline{A}$ = 1
De Morgans	$\overline{AB}$ = $\overline{A}$+$\overline{B}$	$\overline{A+B}$ = $\overline{A}$($\overline{B}$)
		$\overline{A}$+AC = $\overline{A}$+C

common Gates

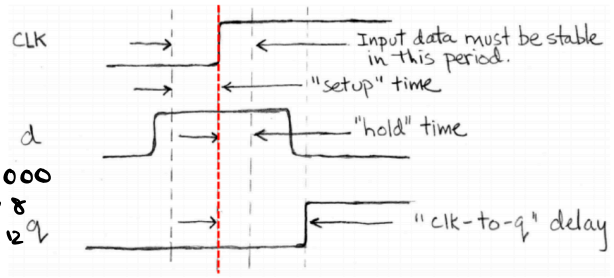


!A!B!C + !ABC + ABC
!AB(C+!C) + ABC
!AB + ABC
B(!A+AC) * Absorption
B(!A+C)

XOR

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

x11 = 0xFFFFFB13
sw x11 0(x5) → x5 = FF FF FB 13
sh x11 2(x5) → x5 = FB 13 FB 13 (a+3 a+2 a+1 a)
lb x12 1(x5) * lb=sign extend → x5 = 0xFFFFFFFB
load byte signed
lbu x13 2(x5) * lbu=0 extend → x13 = 0x00000013
load byte unsigned
srai : divide by 2, round down



TIMING

setup: time before clock edge that input must be stable
Hold: time after clock edge that input must be stable
clk to Q: time after clock edge that input appears on Q
critical path: longest path delay between 2 state elements
path delay: clk-to-Q (FIRST element) + combinational logic + setup time (2nd element)

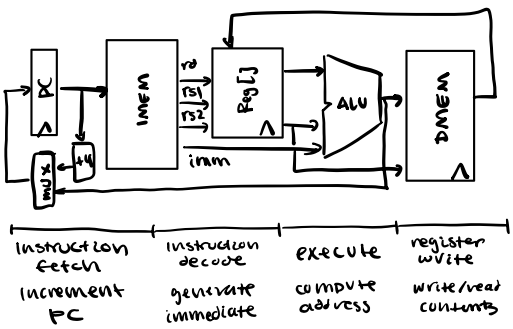
lui : lower 12 bits = 0
upper 20 = instruction type imm
lui t0 0xABCDE : t0 = 0x123 45 000
addi t0 t0 0x678 : t0 = 0x12345678
auipc x5 0x0300 : x5 = PC + 0x3000cc12
PC = 0x40000000 : x5 = 0x40000000 + 0x03000000 = 0x43000000
jalr x0 0xc00(x5) : jump to x5 + sign extend 0xc00 + jump to address

min clock period = critical path
max clock freq = $\frac{1}{\text{min clk period}}$

MAX $t_{clk\text{-}to\text{-}q} + t_{shortest\text{-}comb\ path} \geq t_{hold}$
MIN $t_{clk\text{-}to\text{-}q} + t_{longest\text{-}comb\ path} + t_{setup} \leq t_{clk\ period}$

longer hold = value has potential to change
shorter period = values may not finish computing

DATAPATH



Instr	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4 0	*	*	Reg 0	Reg 0	Add	Read 0	write 1	ALU 1
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm 1	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Write 1	no write 0	*
sw	*	*	+4	S	*	Reg	Imm	Add	write	0	*
beq	0	*	+4	B	*	PC 1	Imm	Add	Read	0	*
beq	1	*	ALU 1	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	signed 0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	unsigned 1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	2 PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

* longest instruction = load instructions

PCSel: 0 = PC+4, 1 = branch+jump

ImmSel = type of immediate

ASel: 0 = register, 1 = PC (eg jalr) — input for A ALU

BSel: 0 = register, 1 = immediate — input for B ALU

ALUSel:
0 = add, 1 = sll, 2 = slt, 3 = unused, 4 = xor, 5 = or, 6 = and, 7 = mul, 8 = mulh, 9 = unused, 10 = sub, 11 = mulhu, 12 = sra, 13 = unused, 14 = bsel

BrUn: 0 = signed, 1 = unsigned
BrEq: 0 : A != B, 1 : A == B
BrLt: 0 = A ≥ B, 1 = A < B
MemRW: 0 = no write, but read; 1 = write DataW @addr (stores s[b/h/w]) ← write to DMEM
RegWEn: 0 = No writes back to RegFile allowed, 1 = writes allowed
WBSel: 0 = mem data to register (loads l[b/bu/h/hu/w]), 1 = ALU output to register (arithmetic), 2 = PC+4 to register (ra = destination register)

STRUCTURAL HAZARD: same piece of hardware being accessed by MORE INSTRUCTIONS than access points
↳ SOLNS: stall / add more hardware, separate IMEM & DMEM [can it be solved by adding another regfile?]

DATA HAZARD: data dependency → data needed by following instr before its updated
ex reading register before prev finishes writing
↳ SOLN: stall / nop, forwarding

CONTROL HAZARD: flow of execution has more than 1 option
* JUMPS + BRANCHES
↳ SOLN: stall, branch prediction, flushing

PIPELINING

- max efficiency in n-stage pipeline = n instructions in pipeline
* add register for every wire carrying value across stages

latency = time for CPU to execute 1 instr
throughput = # of instructions executed per unit time

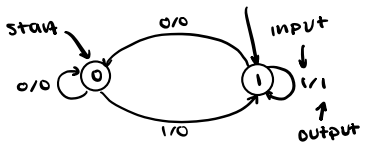
processor performance

$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instruction}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycles}}$

	SINGLE	N-STAGE
clock cycle	full datapath clock per	max(clock per of any stage)
latency	clock cycle	clock cycle * N
throughput	$\frac{1}{\text{clock per of full datapath}}$	$\frac{1}{\max(\text{clock per of any stage})}$

FSM = represented by states + transition

transition 1→1 output 1



DLP = data level parallelism

*optimize $\frac{\text{instruction}}{\text{program}}$ → increase $\frac{\text{work}}{\text{instruction}}$

SIMD = single instr, multiple data
vectorized calculations

notation: _mm_<intrin_op>_<suffix>
- <intrin_op>: basic operation
- <suffix>: data type (packed (p), extended packed (ep), scalar (s))
- return type: __m<size>

* remember edge case for elements that don't fit in vector

AMDAHL'S LAW

$$\text{speedup} = \frac{1}{(1-F) + \frac{F}{S}}$$

F = fraction of program that can be optimized
S = speedup factor for how much that fraction can be optimized

OPENMP

```
#pragma omp parallel
{
// fun stuff
}
```
- copied & executed across all threads INDEPENDENTLY
- if around a loop: copies + executes same loop across many threads

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    // fun stuff
}
```
- SPLIT across # of threads
  - each thread is independent
- scheduled by OS

PLP: process level parallelism

program on multiple processes @ once
manager-worker framework
- manager assigns work
- every other process (workers) ask for work

OPENMPI

```
int main (int argc, char** argv) {
    //set up openMPI
    if (manager process) {
        //manager node code
    }
    else {
        // worker node code
    }
    //terminate openMPI
}
```

```
int main(int argc, char** argv){
int numTasks = atoi(argv[1]); //read in from command line
MPI_Init(&argc, &argv);
// get process ID of this process and total number of processes
int procID, totalProcs;
MPI_Comm_Size(MPI_COMM_WORLD, &totalProcs); //total # of Processes
MPI_Comm_rank(MPI_COMM_WORLD, &procID); //Current Process ID

if (procID==0){
  //manager node code
} else {
  //worker node code
}
MPI_finalize();
```

While work to do:
- wait until worker ready
- find next task
- send to worker what task to do

repeat # worker times
- wait until worker ready for work (recv from all)
- send to worker "all work done"

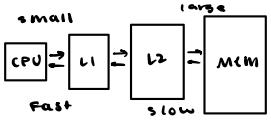
While True:
- send to manger "I'm ready for more work"
- receive manager msg
- If msg = "Here's more work" do the work
- Elif msg = "all work done" break

caches

= small memory storage w/ FASTER ACCESS than DMEM

Logic:
Need to access memory → check cache
- if not in cache → grab from main mem & load into cache
- if in cache → grab from cache

Locality: principle for saving/updating data in cache
1. Temporal: keep most recently accessed items
2. Spatial: move contiguous blocks of memory into cache



DIRECT MAPPED

TAG	INDEX	OFFSET

- TAG: distinguish between all main memory address
- INDEX: specifies which block of cache
- OFFSET: which byte within cache block

# of bits

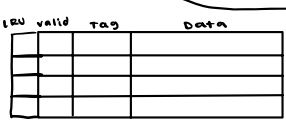
index = log(# blocks / associativity) = $\log_2$(# of indices)
offset = $\log_2$(size of blocks)
tag = (# bits in mem addr) − (# of index bits) − (# of offset bits)
cache size = # of blocks * block size

FULLY ASSOCIATIVE

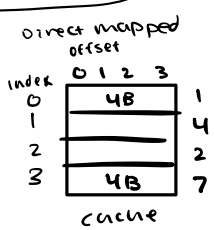
TAG	OFFSET

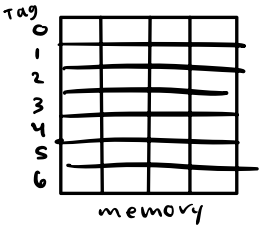
no index bits → can be written anywhere
* NO CONFLICT MISSES
* need comparator for every entry to check valid

cache writes (cache = $, main mem = MM)
1. write through: on every write → data written to $ + MM
2. no write allocate: $ line NOT written to cache → directly to MM
3. write back: data written to $ and DIRTY BIT = 1 → written to MM in eviction
4. write allocate: $ line pulled to $ and written to in specified write-hit policy



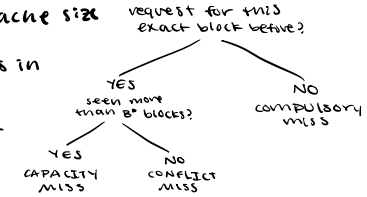
cache eviction policies
LRU = least recently used
MRU = most recently used
FIFO = first in first out
LIFO = last in first out





cache misses
1. compulsory miss: never tried to pull block into cache (ALWAYS happens first time we request)
   - SOLN: increase cache BLOCK size, prefetching
2. conflict miss: had to evict block bc another block REPLACED it and CACHE NOT FULL
   * DOES NOT exist for fully associative
   - SOLN: increase associativity, improve replacement policy
3. capacity miss: evicted block bc cache was FULL
   - increase cache size

B* = # total blocks in cache = $\frac{\text{cache size}}{\text{block size}}$



to = 01111
sll i = 01111000000
srli = 01111000000000000

to = 0xF000 0000
t1 = 0xF000 0000
to = 0x0F00 0000
t1 = 0xFF00 0000

example: summing vals ≥ 128

```
__m128i _127 = _mm_set1_epi(127);
__m128i sum_vec = _mm_setzero_si128();
for (int i=0; i < NUM_ELEMS/4 *4 ; i+=4){
    __m128i tmp = _mm_loadu_si128((__m128i *)(vals+i)) <- load 4 vals
    __m128i comp_vec = _mm_cmpgt_epi32(tmp, _127); <- check vals > 127
    __m128i mask = _mm_and_si128(tmp, comp_vec); <- bitwise AND
    sum_vec = _mm_add_epi32(sum_vec, mask_vals);
}
int tmp_arr[4];
_mm_storeu_si128((__m128i *)tmp_arr, sum_vec);
result += tmp_arr[0] + tmp_arr[1] + tmp_arr[2] + tmp_arr[3];
for (int i = NUM_ELEMS/4*4; i< num_elems; i++){
    if (vals[i] >= 128){
        result += vals[i];
    }
}
```

pointer math

__m128i _mm_set1_epi32(int i)
set 4 signed 32-bit ints to i

__m128i _mm_loadu_si128(__m128i *p)
load into pointed to by p into vector

__m128i _mm_mullo_epi32(__m128i a, __m128i b)
$(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$

__m128i _mm_add_epi32(__m128i a, __m128i b)
$(a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)$

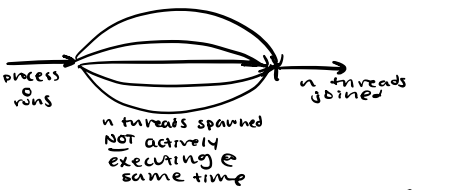
void _mm_storeu_si128(__m128i *p, __m128i a)
store 128 vector at pointer p

__m128i _mm_and_si128(__m128i a, __m128 b)
Bitwise AND of 128 bits in a and b

__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)
ith element of return vector set of 0xFFFFFFFF if ith element of a and b = → o/w 0

* multithreaded CANNOT assume order threads are executed in

fork(): master → child threads & divides work
join(): wait until threads finish → synchronize → terminate all except master



kilo = 10^3 kibi = 2^{10}
$10^3 \approx 2^{10}$ mega = 10^6 mebi = $2^{10\cdot2}$

TLP = thread level parallelism

program = sequence of instrs
process = execution of program
- ↳ each w/ own address space + resources
- ↳ same code but SEPARATE

thread = segment of process
- ↳ same code, same time
- ↳ shared components: heap, global variables
  - INDIVIDUAL registers, PC, stack
- ↳ single core can execute 1 thread at a time

DATA RACE = $\frac{\text{multiple threads modifying same}}{\text{SHARED VAR}}$
- ↳ soln: atomic operations + locks

LOCK: only 1 thread can have lock @ a time
- acquire: try to get lock + keep trying
- release: unlocks lock + continues
- try-acquire: try to get lock but if used → return FALSE

CRITICAL sections: code surrounded by lock → only ONE thread allowed to run it

REDUCTIONS: increase parallelizability by reducing code in critical section

AMAT = average memory access time

hit time + miss rate (miss penalty)
- hit time: time to get data if in cache already
- miss rate: $\frac{\#\text{ misses}}{\#\text{ accesses in a level}}$
- miss penalty: time to get data if it is not in the cache

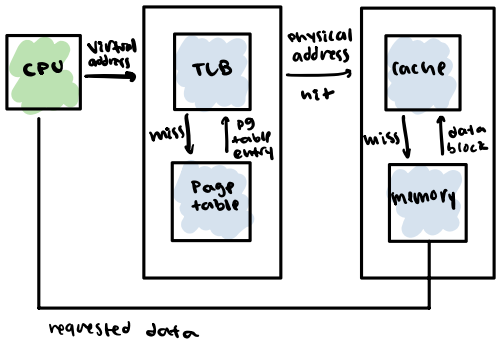
L2 AMAT = L1 HT + L1 MR (L2 HT + L2 MR (main mem access))

cache code analysis
1. count access per inner & outer loop
2. look for compulsory misses, thrashing (going back and forth between filling cache block) and count hits + misses
3. Hit rate = $\frac{\text{hits}}{\text{hits + misses}}$

virtual memory = give programs illusion of more memory

→ protection + security
→ combine disk + memory into larger space
→ allows programs to "see" more memory than exists

pages = uniformly sized block of data
↳ **page table** maps between virtual and physical



offset bits = $\log_2$ (page byte size)
dirty = 1 when write

virtual address

virtual pg # (VPN)	page offset

physical address

physical pg # (VPN)	page offset

TLB = translation lookaside buffer
* cache for page table
* usually fully associative
* RESET when new process started

0xB055ADE1
0xC160EFFF
0x7890ABCD
0x97543210
<truncated>

page table

0x 00000 445
VPN offset

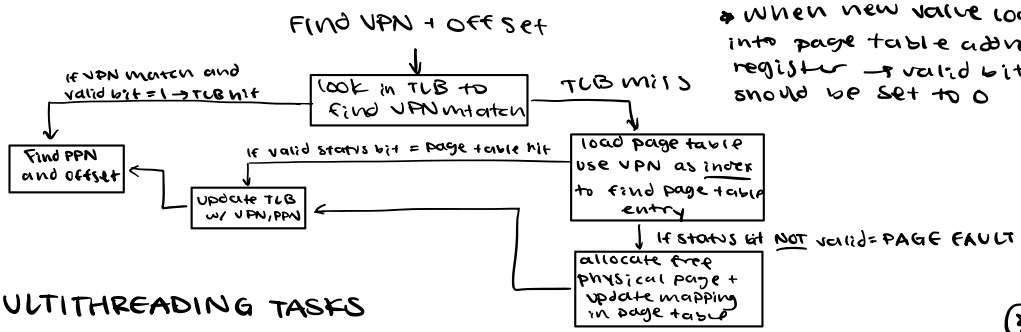
index 0 = 0xB055ADE1
PPN

PPN+offset = ADE1445
MSB = 1 so valid

STEPS

① check if VPN in TLB
↳ check valid bit
1 = hit
0 = check page table

② check page table
↳ check valid bit
1 = hit (NO page fault)
0 = page fault
↓
go to next physical address



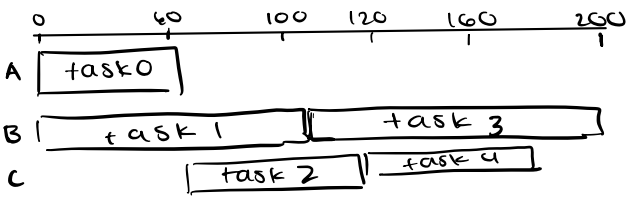
* when new value loaded into page table address register → valid bits should be set to 0

** FOR RISC + C coding questions :

② can use bitwise AND (&) to check if a digit exists with another digit
③ can use slli to multiply
④ malloc(sizeof(int*) * (n))
⑤ calloc(num_elems, sizeof(int))
↑
1 elem

MULTITHREADING TASKS

* find combo of tasks to maximize parallelism (multiple chips doing work @ same time) and chip specialization (chip that's good @ memory doing tasks more demanding on memory)



time = 200

* take greedy approach