Number Representation

Big Idea: Bits can represent anything

N bits $\rightarrow 2^N$ representations

Number Bases: Decimal, Octal, Hexadecimal, etc...

Converting other bases from base 10: "leftover algorithm"

ex) $73_3 \rightarrow$ base 4?

1) $4^5 = 1024 \rightarrow$ fits 0 $\rightarrow$ leftover 73
2) $4^4 = 256 \rightarrow$ fits 0 $\rightarrow$ leftover 73
3) $4^3 = 64 \rightarrow$ fits 1 $\rightarrow$ leftover 9
4) $4^2 = 16 \rightarrow$ fits 0 $\rightarrow$ leftover 9

... $\rightarrow 0\times4^5 + 0\times4^4 + 1\times4^3 + 0\times4^2 + 2\times4^1 + 1\times4^0$

$\rightarrow 73_{10} = 00102 1_4$

Common bases: Decimal, Binary, Hexadecimal

Notation: No prefix, 0b..., 0x...

Binary $\leftrightarrow$ Hexadecimal Conversion: Use the table!

Why it works: each 4 binary digits corresponds to 1 hex digit.

Units: Byte (8 bits, 2 hex), Nibble (4 bits, 1 hex)

Integer Representation: Signed & Unsigned

Unsigned: N bits → $[0, 2^N - 1]$

↳ Overflow/Negative Overflow (not enough digits!)

Signed: Sign-Magnitude → leftmost bit is the sign.

↳ $[-2^{N-1} + 1, 2^{N-1} - 1]$, counting in bits is weird.

Signed: One's Complement → negative, then flip the bits.

↳ always increasing, but overflow on the edges.

Signed: Two's Complement → shift all negatives to left 1.

↳ fixes double representation of zero

↳ Another definition: MSB is now negative

Signed: Bias Notation → shift everything to prevent overflow

↳ "standard bias" centers at zero, math is hard.

Pointers, Arrays, & Strings

*In C, false is only 0 or null pointers (all else is true)

"Memory is just a huge array" → each value has address

Pointer: the value is an address of another variable

↳ if p holds x's address, "p is a pointer to x"

ex) `int *p; int x = 3;`

`p = &x;` ← (&) is the "address of" operator

`print(*p);` ← (*) is the dereference (prints 3)

`*p = 5;` ← able to write on the address

C passes by values, but pointers can give references!

ex)
```
void addOne(int *p) {
    *p = *p + 1;
}
```
← changes the content that p points to, not p

The NULL pointer: pointer to all 0s, no r/w

↳ guard pointers with `if(!p){...}`

Pointer to Structs: `(*ptr).x` ⇔ `ptr->x`

Arrays: int arr[2]; arr[num] is actually pointer

a[i] $\iff$ *(a+i) !! (i automatically scales with type)

↳ pointer + n == pointer + n x (sizeOf(*pointer))

A pointer to a pointer is a "handle" (int **h)

↳ this allows manipulation of pointer values out of scope!

Arrays lose its size information when passed

Strings: array of chars that ends with '\0'

Memory Management

"word size": # of bytes in an address

Endianness: little endian = LSB is stored first

big endian = MSB is stored first

"word alignment": 4-byte boundaries for multiple-byte data

↳ uses padding for structs/small types to enforce alignment

sizeof(): = gives size in bytes (of type or variable)

C Program Address Space: 4 regions

1) Stack: local variable, grows down
2) Heap: requested via malloc(), grows up
3) (Static) Data: variables declared outside main
4) Text (code): program executable loaded, does not change

Global vars → Data, Local vars → Stack, auto freed

Stack: a new frame is allocated every time a function is called
↳ frame stores: return address, arguments, local variable space
The stack pointer tracks the last frame relevant
⇒ locally declared variable are lost when function closes
↳ Don't return a pointer to something on the stack!!!

Heap: "Dynamic" memory that can be alloced, resized, and freed
↳ Huge pool of memory, but not allocated contiguously
malloc(): allocates raw memory, uninitialized from heap
free(): frees memory from heap
realloc(): resizes previously allocated heap block to new size

- `void *malloc(size_t n)` → unsigned int for byte counting
  - ↳ returns a general pointer
  - → might return NULL if out of memory

⚠ Always check for NULL malloc returns ⚠
↳ `if(ptr == NULL){...}`

To allocate a struct:

```
SomeStruct *sp = (SomeStruct*) malloc(sizeof(SS));
```

To allocate an array of 20 ints:

```
int *ptr = (int*) malloc(sizeof(int) * 20);
```

(this depends on the architecture!)

- `free(void *ptr)` → the exact pointer returned by (re)alloc()
  - ↳ always should be manually called for each allocation

- `realloc(void *ptr, size_t size)` → new size wanted
  - ↳ returns a new address of the memory block
    (the data might have been copied to a new memory space!)

Memory Leak: failure to free() allocated memory
Use after free: referencing a pointer after free()
Double-free: trying to free() a memory already freed
(realloc() can produce dangling references if multiple pointers point to the same memory)

Function Pointers

int (*fn)(void*, void*) = &foo;

↳ (*fn)(x, y) calls the function

Generics

General-purpose code that updates blocks regardless of types

ex) malloc returns a void pointer to be casted

Invalid to dereference a void pointer!

Generic memory copying: memcopy(), memmove()

↳ copy does not check overlap, move always makes a temp array.

a char is always 1 byte → use to store a general length array

ex) swap_ends (void *arr, size_t nelems, size_t nbytes) {
    swap (arr, (char *c) arr + (nelems-1)*nbytes, nbytes)
}

↳ pointer arithmetics in 1 incrementals!

Floating Points

Binary Point: boundary between integer and fractional part

$X X . y y$ a "fixed binary point" has a predetermined # of integer and fractional bit.

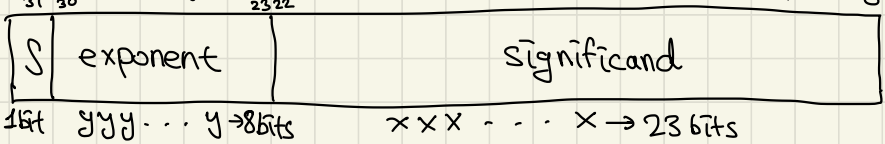
$2^1\ 2^0\ 2^{-1}\ 2^{-2}$

Floating Point: some of the bits carry the binary point location!

Scientific Notation: no leading zeroes → chang to base 2

ex) $1.01 \times 2^{(-1)}$ (mantissa: 1.01, b-point; 2: radix; (-1): exponent) → when normalized, first digit is 1!

↳ Normalized format: $1.XXX\ldots \times 2^{yyy}$ → IEEE 754!



↳ Over/Underflow if the exponent is too big/small

How to store the exponent → bias notation! (exp - 127)

$\Rightarrow (-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}$

Special Numbers: $\pm\infty$ → most positive exponent reserved for ∞ & zero significand

0 → all zeroes in exponent & significand

Max exponent, nonzero significand → NaN representation

exponent 0, nonzero significand → DeNormed numbers

RISC-V

Assembly: set of instructions, operators & operands

Instruction Set Architecture (ISA) ex) RISC-V, x86, ...

Reduced Instruction Set Computer (RISC): small, simple, fast

One line of assembly code $\Longrightarrow$ One instruction

No type, bits in registers get interpreted by operators

Registers are inside the processor, interaction is very fast

RISC-V has 32 registers that are 32 bits wide

↳ 32 bits is 1 word, 32 = word size (4 bytes)

x0 is special and is always 0. (editing is impossible)

RISC Syntax: opname rd, rs1, rs2

ex) add x1 x2 x3 $\Longleftrightarrow$ a = b + c (in C)

Using temporary registers is possible, but might want to minimize

Immediates: numerical constants! ex) addi x_3 x_4 10

↳ the last operand must be a number instead of a register

RISC-V Data Transfer

How do we load from and store to memory?

↳ interacting w/ memory is slow... (later on improvement)

data flow ←

lw (Load word "from") rd, [byte offset](rs) → pointer to array

sw (Store word "to") rd, [byte offset](rs) →

* rs + (byte offset) must be a multiple of 4 for integers!

lb, sb: load and store byte into the low byte position, → LSB

and extends the leftmost bit to the rest of the word

↳ endianness matters in this case! (lbu for unsigned, no extend)

lbu extends all zeroes for "unsigned byte"

RISC-V Procedures

When calling a function...

① Put arguments ② Transfer control to function ③ Acquire storage

④ Do the function ⑤ Store the return value ⑥ Transfer control to callee

Calling Conventions: what registers do what job

a0-a7: argument registers (a0-a1 are return registers as well)

ra: return address to return to point of origin

s0-s1, s2-s11: saved registers

Instruction Support for Functions:

Every line of code also lives in memory like data.

When a function is called, the next line's address is stored to ra.

PC jumps to the address stored in the j (jump) call.

After the function terminates, it returns via jr ra (jump to register).

* jal (jump and link) removes hardcoding line numbers!

↳ jal rd, FunctionLabel saves next line's address to rd and jumps to the address associated with the FunctionLabel

↳ jr ra is also aliased as "ret"

(jalr rd, rs, imm can manually set PC to imm(rs))

When a function is called, old register values are stored and then restored just before returning ⇒ use stack!!

push the sp for space, pop the sp to free stack space

Nested Calls?? → clobbers values in a0-a7 & ra

↳ the nested function will overwrite the old return address!

⇒ save ra to the stack, too

Register Convention: rules for what registers may be altered

1) Preserved across function calls: sp, gp, tp, s0-s11

2) Not preserved: a0-a7, ra, t0-t6 ↳ the callee is responsible for preserving these

RISC-V Instruction Format

PC (Program Counter) is a pointer for instruction count

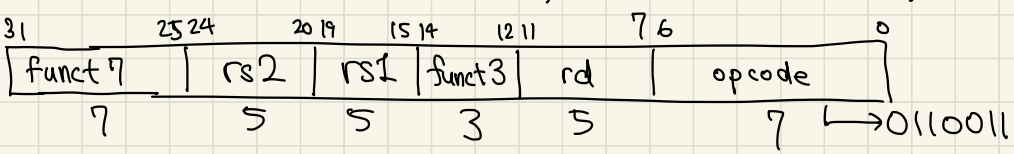
Assembly code ⟹ Machine Code (binary)

Most data are in words (32-bit), RISC-V uses 32-bit.

Fields in an instruction tells something meaningful

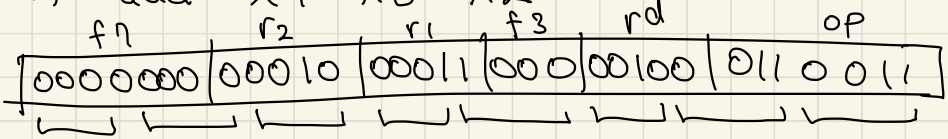
6 Key Instruction Formats: R, I, S, B, U, J

R-Format: opname rd, rs1, rs2 (add, sll, ...)



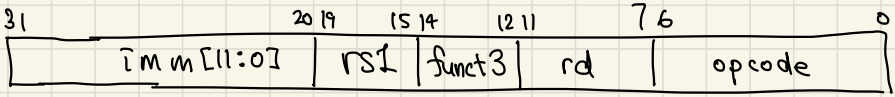
opcode "partially" specifies the instruction, and funct3&7 describes specifics of which version of the instruction to run

ex) add x4 x3 x2



$(0\ 0\ 2\ 1\ 8\ 2\ 3\ 3)_{16}$

I-Format: opname rd, rs1, imm (addi, slli, ...)

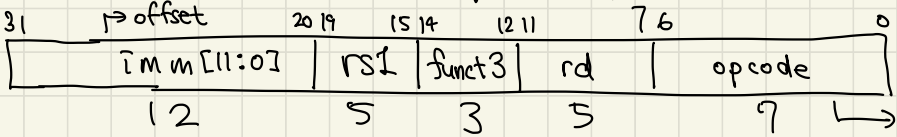


↳ $[-2048_{10}, 2047_{10}]$, CPU sign extends before operating

shift operations use only 5 bits for shamt (shift amount)

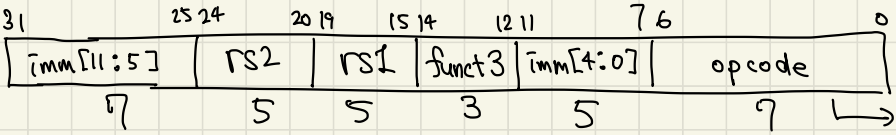
↳ the leftover bits tells logical / arithmetic

I-Format (Load): loadop rd, imm(rs1)



funct3 specifies how much to load in what format (lw, lb, ...)

S-Format: storeop rs2, imm(rs1) → no change in register!



the immediate is split into two parts (RISC-V prioritizes keeping register fields in place more than immediates)

Program Counter (PC): pointer to place in code

↳ +4 after most lines, updated to a label for jumps

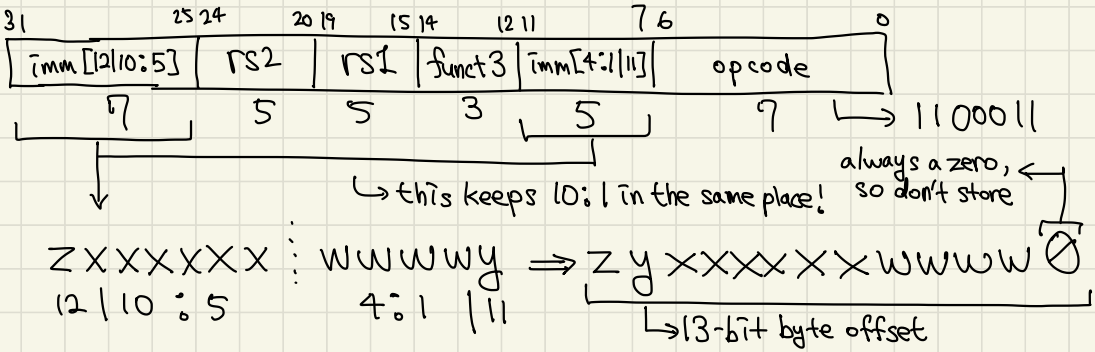
PC-Relative Addressing: PC = PC + (byte offset)

↳ this enables Position Independent Code (no hardcoding)

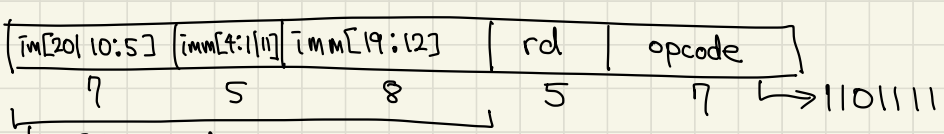
B-Format: branchop rs1, rs2, Label → relative byte offset to the label

↳ label only gets 12 bits → In units of 2 bytes (half-words)

⇒ PC relative offsets support [-4096, 4095]

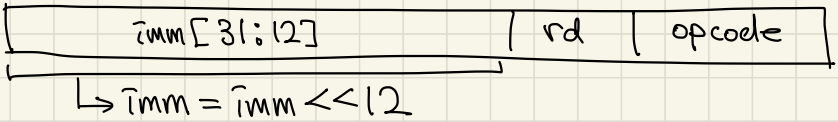


J-Format: jal rd, Label



U-Format: opname rd, immed

↳ "upper immediate"



lui rd, immed: load immed to upper 20 bits, clear rest

↳ then use addi to load the lower 12 bits!

(li pseudoinstruction handles lui + addi implicitly)

Edge Case: addi sign extends and decrements next bit

↳ if 12-bit immediate is negative, add 1 to upper immediate!

auipc rd, imm → rd = PC + (immed << 12)

(auipc rd, 0 stores the current address to rd!)

jalr rd, rs1, imm → PC = rs1 + imm, rd = PC + 4

↳ this is just an I-Format instruction (imm not x2)

⇒ lui + jalr can access all 32-bit address

CALL

How to translate high-level code to machine code?

Compiler: High Level Language → Assembly Code

↳ output may include pseudoinstructions (mv, li, j, ...)

Assembler: Assembly Code ⟶ Machine Language Module

↳ object code & information for linking & debugging

replaces pseudo instructions with real instructions!

Directives inform how to build object files!

↳ .text, .data, .globl, .string, .word

Object File Format:

1) Header: size & position of other parts of the file
2) Text Segment: machine code
3) Data Segment: static data in machine code
4) Symbol Table: store undetermined absolute addresses
5) Relocation Information store labels for other files to reference for the linker to resolve (To-do list)
6) Debugging Information

Linker: Object files ⟶ executable machine code

↳ enables separate compilation of files!

Patches text and data segments, then resolve references

↳ through relocation table

PC-relative addressing don't need relocation!!

External function reference & static data needs relocation

Static vs Dynamic Linking: self-contained vs only references

DLLs saves space but sacrifices runtime overhead

Upgrading libraries is much more reasonable in DLL

Loader: executable code ⟶ running program

Synchronous Digital Systems

Synchronous: All operations coordinated by a central clock

Digital: All values are discretized to 0s or 1s.

switches: open when 0, closed when 1 (asserted)

AND gate: $Z \equiv A$ and B (Z is on only when $A=1$ and $B=1$)

OR gate: $Z \equiv A$ or B (Z is on when $A=1$ or $B=1$)

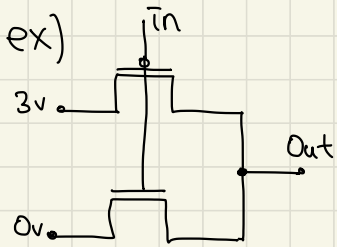
Transistor: MOS transistors act as voltage-controlled switches

n-channel: open when G is low, closed when G is high

S→D flows when G is high enough

P-channel: closed when G is low, closed when G is high
S→D flows when G is low enough

ex)



Relationship between in and out?

in	out
0v	3v
3v	0v

Signals & Waveforms (clock)

↳ Assume transmission is effectively instant

↳ Implies that a wire has only one value at a time

Circuit Delay: Transistors take time to calculate outputs!

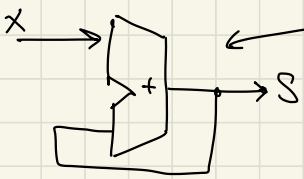
↳ only look at the value after the propagation delay

Types of Circuits: ① Combinational Logic Circuit (CL) ② State Elements

↳ state elements can store information for future calculations!

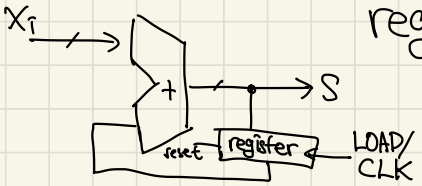
States

Accumulator: $X_i \rightarrow$ [Sum] $\rightarrow S$



← doesn't work because

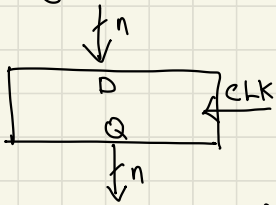
1) no control over loop 2) no initialization



register is used to hold up transfer of data

↳ n-instances of flip-flops

Register:



a rising edge-triggered flip-flop samples the input d and outputs to q on a rising edge

* there is a "clk-to-q delay" and a "hold time"

If input X is not synced with clk, the register may capture a wrong value temporarily, but it is fixed eventually.

⇒ Max delay = CLK to Q delay + Logic delay + Setup time

Finite State Machine: states and transition function

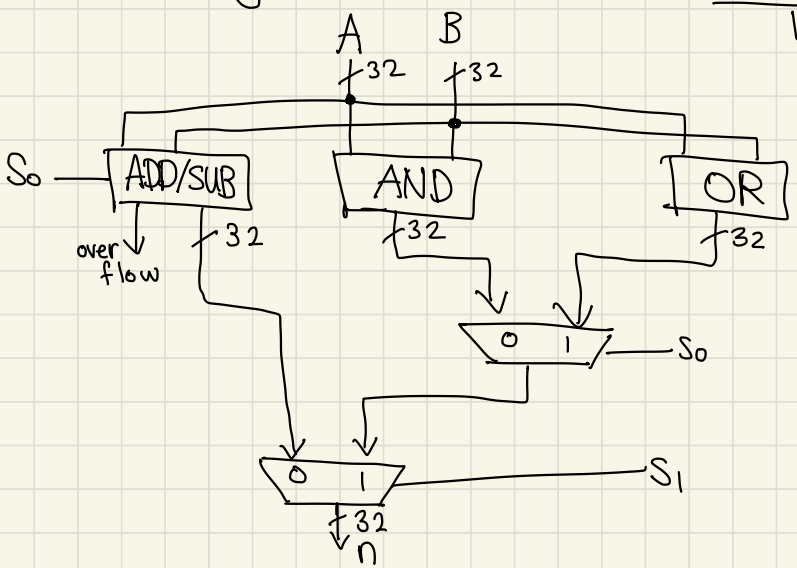
↳ each transition has an input and an output

* FSM can be mapped to hardware with a CL and register!

Combinatorial Logic Block

Multiplexor: signal chooses which input gets outputted

Arithmetic Logic Unit: ADD, SUB, AND, OR
↳ Bitwise



Adder/Subtractor Design: Truth Table or Cascade Layer?

↳ Truth table needs 2^{65} rows...

a_0	b_0	S_0	C_1 (carry bit)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

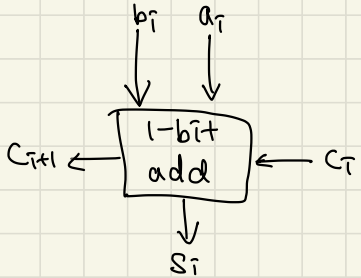
For LSB (without carry input)

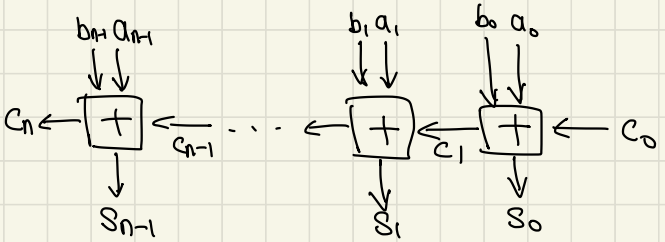
$S_0 = a_0$ XOR b_0, $C_1 = a_0$ AND b_0

Every other row will have a carry bit as an input

↳ $S_i = XOR(a_i, b_i, c_i)$, $C_{i+1} = MAJ(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$

The 1-bit Adder ⟶ N-bit Adder





→ Is $C_n = 1$ necessarily an overflow? Yes, for unsigned.

↳ what about signed numbers? Only when C_n XOR C_{n-1} !!!

Subtractor: Realize that $A - B = A + (-B)$

↳ All bits in B are XORed (→ conditional invertor) with 1, then $C_0 = 1 \Rightarrow (-B)$

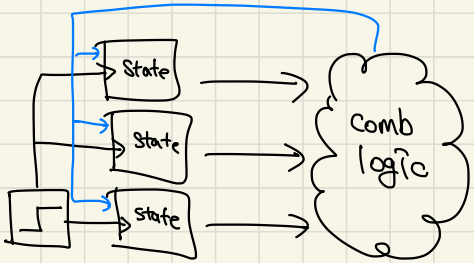
⇒ The SUB signal is connected to XORs to B and C_0 !!!

RISC-V Datapath

Processor → Datapath: the logic of the hardware
Processor → Control: the choicemaking decisions

One-Instruction-Per-Cycle RISC-V Machine

↳ composed of combinational logic blocks & state elements

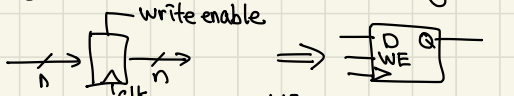


current output of state elements are input to CL, and its outputs affect the state elements on the next clock cycle

States required by RISC-V ISA: PC, Reg[ ], IMEM, DMEM (IMEM → Instruction, DMEM → Data)

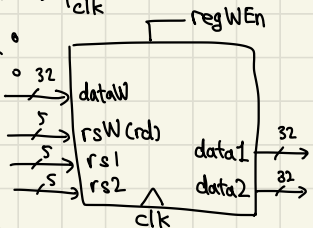
* Every RISC-V instruction changes some state element!

PC:



if WE=1, Q←D on RCE.

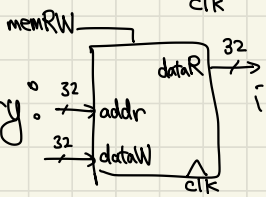
Register:



registers are accessed via their 5-bit register numbers: R[rs1], R[rs2], R[rd]

write operations put dataW into R[rd]!

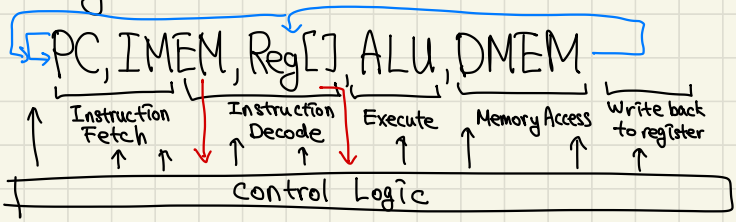
Memory:



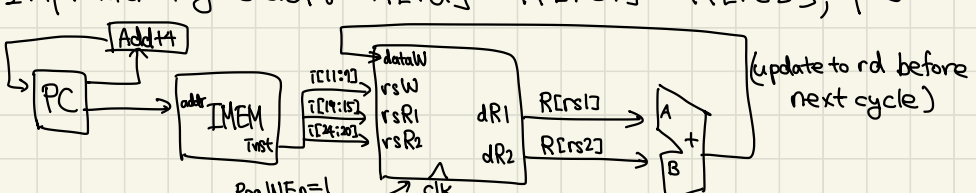
if memRW=1, write occurs. memRW=0, read to dataR.

IMEM removes dataW, clk, & memRW for read only!

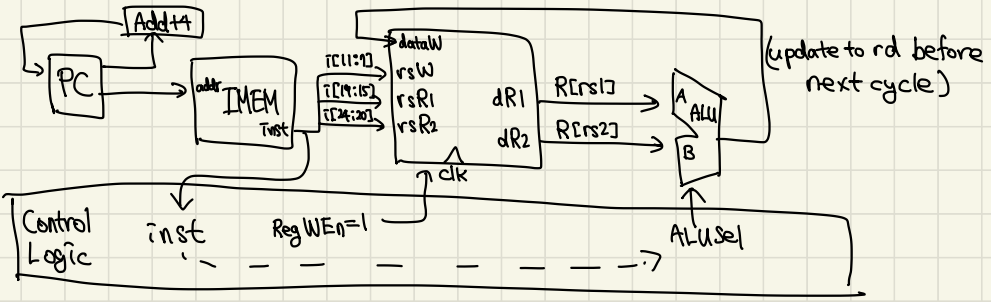
5 Stages of Instruction Execution:



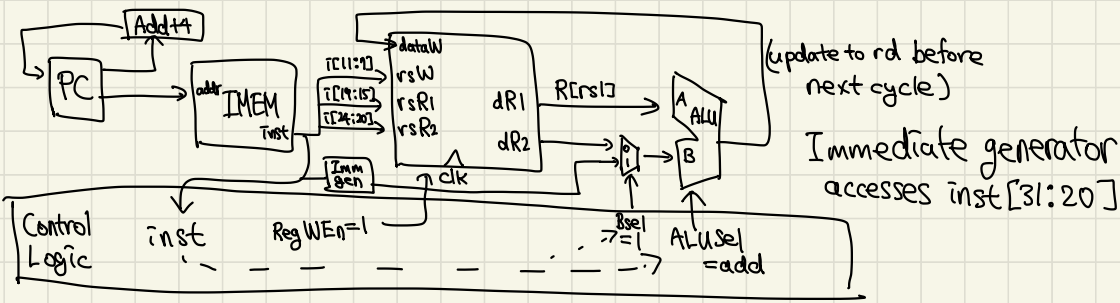
Implementing add: R[rd] = R[rs1] + R[rs2], PC += 4



Implement sub: only differ in one bit in func7, inst[30]!

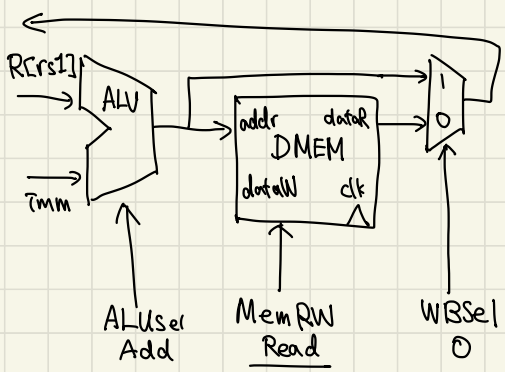


Implement addi: how to take the immediate?

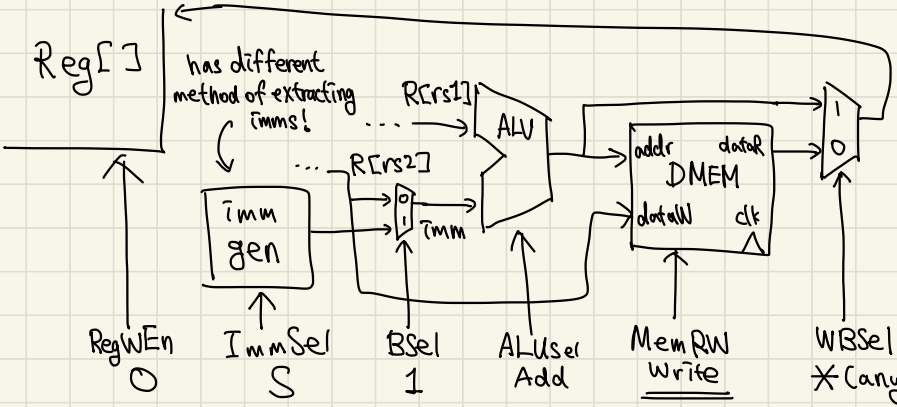


ImmGen: copy to 12 bits, then smear the most upper bit

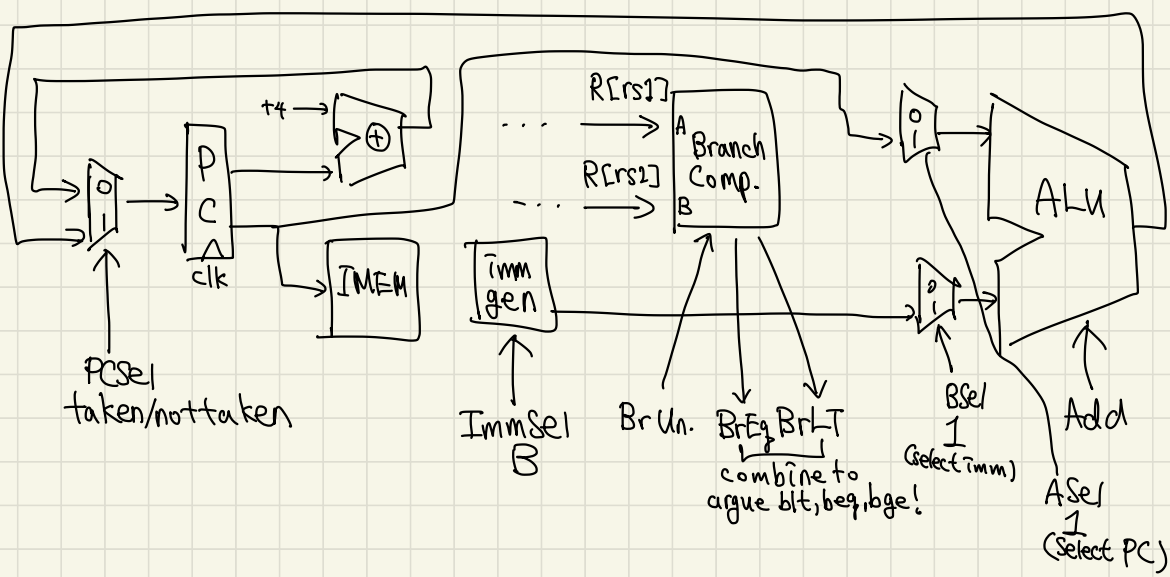
Implement Load: actually similar to addi! with DMEM access now



Implement Store: saves to R[rs2] in memory!



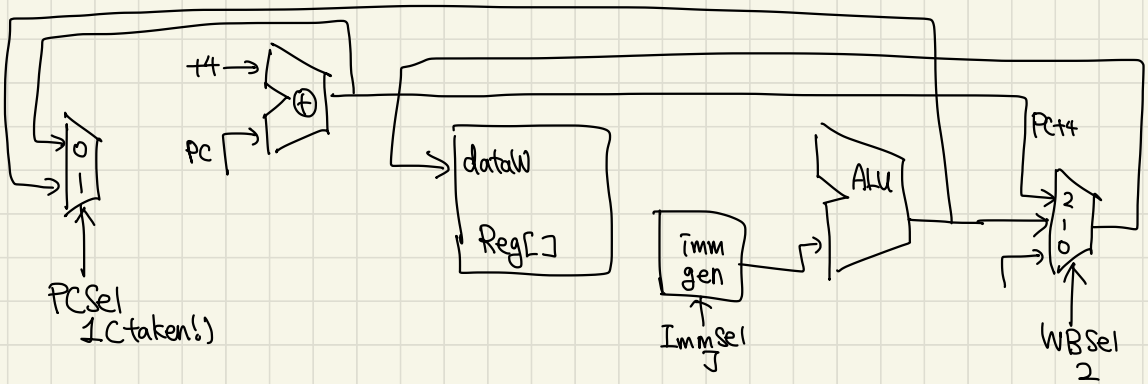
Implement Branches: new imm. format, pc conditionally changes!



Imm. Gen. for B-Types: MUX for imm[1] { S: instr[31] / B: instr[7]

MUX for imm[0] { S: instr[7] / B: 0 (implicit)

Implement jal: update PC, and save ra to R[rd]



Implement jalr: almost same, but starts from R[rs1] (absolute)

↳ just use the I-format, but MUX selects R[rs1] instead of pc

Implement U-Format: Change the Imm.Gen accordingly

LUI → R[rd] = imm, so ALU just selects the Bline!

AUIPC → R[rd] = PC + imm, so ALU just does it.

Datapath Control

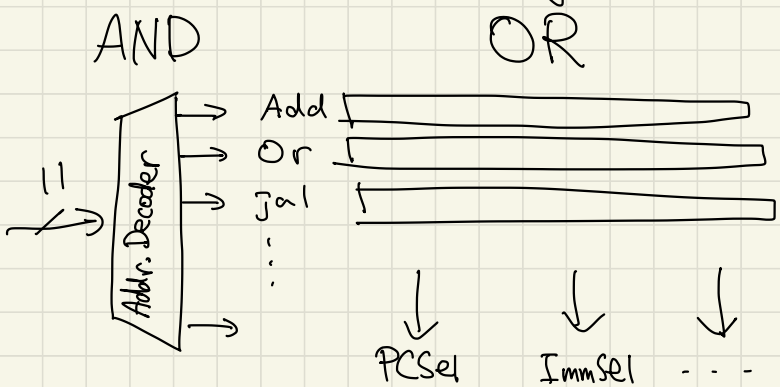
Somehow, we need to extract control logic from instr. bits

Critical Path: $t_{c\text{-}q} + \overbrace{\left[t_{IMEM}}^{T_{IF}} + \overbrace{t_{Reg}}^{T_{ID}} + \overbrace{t_{mux} + t_{ALU}}^{T_{Ex}} + \overbrace{t_{DMEM}}^{T_{MEM}} + \overbrace{t_{mux}\right]}^{T_{WB}} + t_{setup}$

Two ways to realize control: ROM or Comb. Logic

In RISCV32, we only need 11 bits, inst[30, 14:12, 6:2] + BrEq, LT!

ROM Controller: One-hot encoding for all instructions



Pipelining

"How do we improve our datapath performance?"

↳ what does it mean to improve performance?

① Program Execution Time → Time/Program

② Throughput

③ Energy per Task → Energy/Program

$$\frac{\text{Time}}{\text{Program}} = \underbrace{\frac{\text{Instruction}}{\text{Program}}}_{\text{PL, compiler, algorithm, etc.}} \cdot \underbrace{\frac{\text{Cycle}}{\text{Instruction}}}_{\text{ISA \& Datapath}} \cdot \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{Critical Path}}$$

→ Execution depend on all 3!

$$\frac{\text{Task}}{\text{Second}} = \frac{\text{Joule}}{\text{Second}} \cdot \frac{\text{Task}}{\text{Joule}}$$

$$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instruction}}{\text{Program}} \cdot \underbrace{\frac{\text{Energy}}{\text{Instruction}}}_{CV^2\text{, hardware}}$$

C ← capacitance

V ↳ supply voltage

→ reducing supply voltage is hard

Pipelining: make an assembly line-like structure

↳ Latency (single task speed) is unchanged, but throughput increases!

↳ max speedup = # of stages, limited by the slowest stage

⇒ We can apply this to our RISC-V datapath with registers!

↳ clock now tells stage time, not instruction time

However, there are some things to be careful about...

* the register is actually accessed twice (decoding & write back)

↳ the rd is not what we originally meant → send it during WB!

* the control logic needs to be pipelined (either the instruction or control)

Pipelining: Hazards

Structural Hazards: two instructions need the same resource

① Instructions take turns (incurs "stall time")

② Add more hardware ③ Design ISA to avoid conflict (RISC-V!)

Data Hazards: register values don't get updated instantly!

↳ Three solutions: ①Stalling ②Forwarding ③Code Scheduling

①Stalling: insert noop (no instruction) intentionally to delay instructions

↳ the compiler needs to know about the pipeline!

② Forwarding: communicate the computed value earlier than WB

↳ needs extra hardware to implement instantaneous "shouting"

↳ extra wire & extra logic to find whether forwarding is needed

ex) wire the output of ALU to A input of ALU for immediate usage

If the forwarding is still too slow, we need stalling. (Asel now has more choices)

ex) value from lw is wired to the A input of ALU

③ Scheduling: use a no-op slot to execute an unrelated instruction

Control Hazards: branches aren't taken instantaneously!

↳ If we take a branch, cancel some future instructions as no-op

↳ To cancel, force all Write controls to 0 (nothing is updated)

Branch prediction can increase performance on average

More pipeline depth ⇒ Faster, but more potentials for hazards!

Superscalar: Multiple execution units in parallel ⇒ CPI < 1!

↳ $CPI = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Time}}{\text{Cycle}} \right)$

Caching

Binary Prefix: kilobyte = 1024 bytes, in SI Units, 1000 bytes.

↳ Hard Disk & Telecommunications use SI, all else "binary prefixes"

kibi → 2^{10}, mebi → 2^{20}, gibi → 2^{30}, ... ⇒ $2^{XY} = 2^{Y} \cdot 2^{10X}$

Cache: Middle memory between processor and main memory (DRAM)

↳ Secondary memory (disk / flash) is even farther away

* closer to processor ⇒ smaller, faster, expensive, and subset

* Cache stores instructions and data most relevant to program

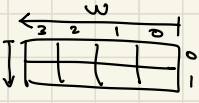
Locality: Temporal & Spatial locality gives direction on what to save

⇒ recently accessed data can be stored close to processor!

⇒ move blocks nearby the referenced data together!

Direct Mapped Caches: memory address mapped to one block in cache

↳ Each block can hold an 2^{x} number of bytes → shift arithmetic!

⇒ In address: $\underbrace{t\,t\cdots t}_{\text{tag}}\,\underbrace{i\,i\cdots i}_{\text{index}}\,\underbrace{0\,0\cdots 0}_{\text{offset}}$ where 

(think of the index & offset like row & column coordinates where # of bytes of index & offset are log(h), log(w)!)

⇒ tag length = addr length − offset − index ✱

Memory Access with Cache: If hit, send. If miss, fetch block.

↳ Cache never writes to memory, only copies it!

Cache miss, block replacement: wrong data, so fetch and overwrite

Cache Temperature: Cold Warming Warm Hot ← → for performance

Miss penalty: time lost by missing a cache and replacing it

Valid Bit: 0 → cache miss/garbage 1 → cache hit if addr.=tag

↳ when initializing a program, set all valid bits to 0

How to handle writing?

↳ Write-through updates both cache and memory

↳ Write-back allows inconsistency (being "stale") and updates when replaced

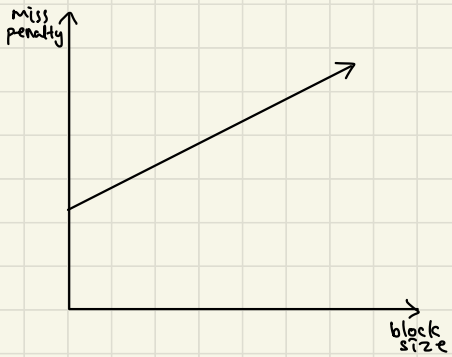
(also need to add a "dirty" bit to remind inconsistency)

⇒ Trade-off between performance and complexity

Is large block size good? There is also a trade-off.

↳ Spatial locality ∝ block size, but also larger miss penalty

↳ an extremely large block size will need to discard data too often (ping-pong effect)



Types of Misses:

1) Compulsory Miss: First time, so it must be empty

2) Non-compulsory Miss: All other misses

Fully Associative Cache: put data on any row, but need to compare against all rows when reading from cache ⇒ infeasible hardware

Set-Associative Cache: A middle-ground for associativity

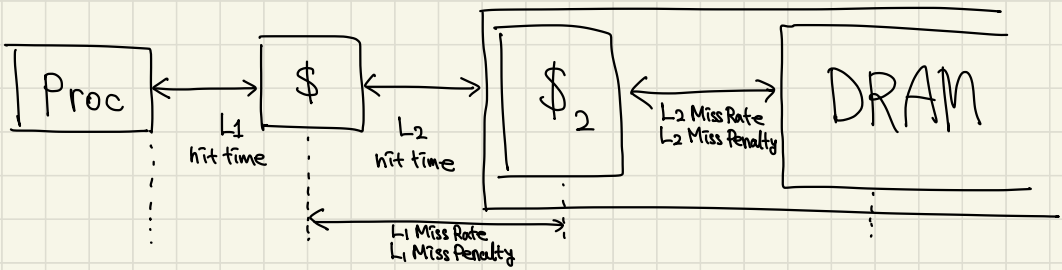
↳ Each set acts like a fully associative cache, but the set itself is directly mapped to the index value ⇒ much more lenient!

Block Replacement Policy: If all valid bits are on in the same set, which to replace with?

ex) Least Recently Used (LRU), FIFO, Random, etc.

Average Memory Access Time = Hit Time + Miss Penalty x Miss Rate ☆

↳ How do we improve the miss penalty? Second Level Cache!!



$AMAT = L_1 \text{ Hit Time} + L_1 \text{ Miss Rate} \times L_1 \text{ Miss Penalty}$

$L_1 \text{ Miss Penalty} = L_2 \text{ Hit Time} + L_2 \text{ Miss Rate} \times L_2 \text{ Miss Penalty}$

$\Rightarrow AMAT = L_1 \text{ Hit Time} + L_1 \text{ Miss Rate} \times (L_2 \text{ Hit Time} + L_2 \text{ Miss Rate} \times L_2 \text{ Miss Penalty})$

→ hardware limit for clock speed

How to improve miss rate? Larger cache, or increase associativity

Parallelism

Single Instruction/Single Data Stream: What we did upto now

Single Instruction/Multiple Data Stream (SIMD): parallel operations!

Multiple Instruction/Multiple Data Stream (MIMD): multicore/WSC

MISD is not relevant anymore (Why would you do this...)

Hardware \ Software	single	multiple
single		
multiple		

SIMD Architectures: Data-Level Parallelism (DLP)

ex) vector element-wise multiplication in a single cycle

Advanced Digital Media Boost: MultiMedia eXtension (MMX)

→ developed to wider registers for more parallelism (512b)

XMM registers: eight 128-bit data registers (packed!)

↳ allows four single-precision operations in parallel

(Intel uses 16 bits for a word, 32 for float, and 64 for double)

SIMD Array Processing: for every 4 members in array...

Intrinsics: C functions & procedures that can choose SSE instructions

↳ How to do this in RISC-V? → add SIMD instructions!

Thread-Level Parallelism

ex) CPU with two cores: two processors executing their own instructions

↳ Separate: Datapath(PC, Reg, ALU), L1&L2 cache

↳ Not separate: Memory (DRAM), L3 cache (not required)

Thread: A single stream of instruction, a program can fork into multiple threads. Single core can execute each thread via time sharing.

Each thread has: a dedicated PC, registers, and access to shared memory

Hard ware thread: ones running on cores, Sofware thread: all else

↳ Multiplex sw thread onto hw threads for efficiency!

Removing a sw thread from hw: Interrupt, save registers & PC

Load a sw thread to hw: Initialize a core with loaded registers & PC

Multithreading: is swapping threads while stalled worth it?

↳ Two copies of PC & registers → executes like two threads!

⇒ More Logical CPUs than Physical CPUs! (Hyper threading)

OpenMP

Parallel Loops: #pragma omp parallel for

↳ code must be resilient to indeterminism in order!

Fork-Join Model: forks and executes simultaneously, then joins

Race Condition: result depends on the parallelized operation order

Synchronization: Limit access to shared resource to 1 actor at a time

↳ use a "lock" to signify possesion of a variable by a thread

⇒ However, there can be a race condition for lock possesion!

Hardware Syncronization → atomic read/write! no interruption in btwn

Atomic Memory Operation: performs operation in place, old value to rd

↳ This enables abstraction to declare a critical block of code

Deadlock: a system state in which no progress is possible

OpenMP timing: double omp_get_time(void) → wall clock time

Shared Memory Multiprocessor: Single memory space shared by all

↳ each processor has their own private cache → incoherent values?

Cache Coherency: notify other processors when a write or miss happens

↳ a shared block is consistent with memory

↳ a modified block is changed, no other cache has a copy, memory out-of-date

↳ an exclusive block is same as modified except memory is up-to-date

↳ an owner block, other caches can have a copy (shared state),
the owner must respond to a snoop request with data.

False sharing: two caches claim that they have the valid block

↳ Introduces a Coherence Miss!

Distributed Computing

"Many different programs working together to achieve common goal"

→ Concurrency is hard: no shared memory/state, locks are hard.

→ Failure handling is hard: "zombie processes" keeps running in one crash

→ Communication is hard: transmission delay, message expectation?

⇒ Split into independent subtasks, minimize communication!

Manager-Worker Framework: One manager only assigns work to others

↳ needs agreement on what instructions are supposed to mean

↳ manager doesn't do work to prevent stalling giving out instructions

MapReduce: Abstraction for jobs involving mapping and reduction

↳ transform via mapping, then group elements by keys for reduction

Virtual Memory

"Give each process the illusion of using their own memory"

→ Physical vs Virtual Addresses, OS translating between

Naive approach: 1-to-1 mapping table for each address

Pages: Map ranges of contiguous memory → saves table columns!

↳ store only the top bits necessary to tell pages → page numbers

ex) Virtual page 0x12345 has address 0x12345000 ~ 0x12345FFF

⇒ In real life, only physical page numbers are stored, and VPNs are indices.

Page tables also lives in memory ⇒ ✱ every load/store is two mem access! ✱

↳ ① Fetch PPN from page table ② Access actual data in memory

Parameters: Page size, VM size, PM size ⇒ affects # of bits

VPN bits = VM bits − offset bits, and so forth.

Size of Page Table: # of entries × size of each entry

What if physical memory < virtual memory? → use disk memory!

↳ accessing data not in memory causes a page fault, evicting another data in memory and updating the page table ↳ PPN is garbage

How to allocate pages? → Demand paging: only load upon request

Writing Data: ① write through ② write back (like cache)

↳ ALL VMs uses write-back (disk is too slow), use dirty bit

⇒ Gives illusion of larger memory and demands isolation btwn programs

... sometimes, multiple program might need the same memory

→ just direct both programs to the same physical page.

A Read-Only bit can be used to protect certain pages!

Translation Lookaside Buffer(TLB): "Caches" for page tables. (→ only one TLB per core)

↳ Fully associative, just remembers the last few pages accessed

TLB Reach = # of TLB entries × Page size (immediate resolve!)

* When the OS context-switches, it must invalidate the TLB (flush!)

⇒ Three cases: ① TLB hit (Best) ② Page Table hit (Not worst) ③ Disk access (Worst)

↳ if TLB or Page Table faults, update it on the way back!

VM + Caches: Physically Indexed, Physically Tagged (PIPT)