RISC-V Instruction

- R-Format : op rd, rs1, rs2
  - Eg. add, xor, mul
- I-Format : op rd, rs1, imm
  - Eg. addi, lw, jalr, slli
  - Shift ops instruction : funct7, shift amount (5-bit) since only shifts b/w 0-31 makes sense, rs1, funct3, rd, opcode
  - Special : load-op rd, imm-offset(rs1)
- S-Format : op rs2, imm-off(rs1)
  - Eg. sw, sb
- SB-Format : op rs1, rs2, offset from PC
  - Eg. beq, bne, blt, bge
  - PC-relative addressing : PC + imm * 2, since imm in half-words = 2 bytes (16 bits). Lowest bit always 0, so no store.
- U-Format : op rd, imm
  - Eg. lui, auipc (different!) with 20-bit upper imm
  - Special : sign extension of lower 12 bits since addi is always sign-extended
    - lui x10, 0xDEADB → x10 = 0xDEADB000
      addi x10, x10, 0xEEF → x10 = 0xDEADAEEF
    - pseudo-op handles this : li x10, 0xDEADBEEF by creating two instructions and pre-incrementing 1 if necessary
- UJ-Format : jal rd, offset from PC (label)
  - Eg. jal
  - Target $\pm 2^{19}$ locations, 2 bytes apart, i.e. $\pm 2^{18}$ 32-bit instructions
  - j Label is pseudo → set rd=x0
  - Special : jalr is I-type, no 2-byte multiplication
    jalr rd, rs1, offset : rd = PC+4 , PC = rs1 + offset
- Branches can't be too far (so no change when combining), while general jumps (jal), we may jump to anywhere in code memory (need change in merging binaries)
- Can always : jalr ra, address you want, 0 (0 is the offset) if ra is saved in calleE

FP, CALL

- FP : 32 bits (31-0), a way of approximation
  - 31 : Sign
  - 30-23 : Exponent (biased to preserve linearity), usually 8-bit so bias $= -(2^8 - 1) = -127$ for normal
  - 22-0 : Significand, implicit 1 for normal
- Normal : $(-1)^{Sign} \cdot (1.\text{Significand}) \cdot 2^{\text{Exponent}-127}$
  - Smallest $= \pm 2^{-126}$
  - Largest $= \pm(2^{128} - 2^{104})$
- Shifting bias of Exponent in FP (-127 in norm $\rightarrow -126$ in denorm) AND Significant has a leading 1 in norm (no leading 1 in denorm)

  Denorm : $(-1)^{Sign} \cdot (0.\text{Significand}) \cdot 2^{-126}$
  - Smallest $= \pm 2^{-149}$
  - Largest $= \pm(2^{-126} - 2^{-149})$
- With denorms, gap $= 2^{-149}$ consistent
- Overflow (abs. value too large) & Underflow (abs. value too small) : might incur rounding errors
- Addition not associative : small + big + small vs. small + small + big (rounding)
- float f, then f == (float) ((int) f) is not always true (especially for < 1 or non-int floats)

  int i, then i == (int) ((float) i) not always true, since large ints don't have exact FP reprs; but if use double, since significant is 52 bits $>$ 32 bits of int, so always true

Exponent	Significand	Meaning
0x00	0	± 0
0x00	non-zero	± Denorm Num
0x01 – 0xFE	anything	± Norm Num
0xFF	0	± ∞
0xFF	non-zero	NaN

- Compiler, Assembler, Linker, Loader
  1. Compiler : .c to .s (can include pseudo-ops)

2. Assembler : .s to .o (object code, machine language, true assembly)
   - Directives : .text, .data, .word
   - 2 passes over the program for forward references
   - Jumps to external labels (?) and references to 32-bit address of data can't be determined, so needs 2 tables
     * Symbol Table (ST) : list of "items" may be used by others, i.e. labels (function calling) & data (.data)
     * Relocation Table (RT) : list of "items" this file will need, i.e. external label jumped to (jal/r) & data (e.g. ref. in .data)
   - Object File Format
     (a) object file header : size and position of other pieces
     (b) text segment : machine code
     (c) data segment : data in source file (binary)
     (d) RT : identifies lines of code that need to be "handled"
     (e) ST : list of this file's labels and data that can be ref-ed
     (f) debugging info
3. Linker : .o to .out (executable)
   - Enables separate compilation of .o object files and then combine into a single .out executable (linking)
   - Combines .text, combines .data, and then resolve references (go through RT and fill in all absolute addresses)
   - 3 types of addresses
     * PC-relative (beq, bne, jal) : never relocate
     * External function ref (jal) : always relocate
     * Static data ref (auipc, addi) : always relocate
   - For RV32, Linker assumes first word of first text segment is at 0x10000
4. Loader : .out to memory (program is run), = OS usually

Control Logic

- NMOS gate = 1 or PMOS (with a circle) gate = 0, then switch is closed (connected)
- AND : dome-shape; OR : $\sim$ triangular; XOR : or with another curve
- Sum of Products (SoP) : take rows with 1's in output, and take their sum
- Product of Sums (PoS) : take rows with 0's in output, and take their product
- $\neg AB + A\neg B \equiv (A + B)(\neg A + \neg B)$
- Register is state element (for storing & control flow)
- Critical path : longest path (delay) from register to register
- Setup time : amount of time before the clock rise that the input signal must remain constant
- Clk-Q delay : time after the clock rise that the output signal takes to reflect the new value
- Hold time : time after clock rise that input signal must remain constant
- critical path = max delay = min period (assuming hold time $<$ max delay) = clk-Q + longest Combination Logic + setup
- clk-Q + critical path + setup $\leq$ clock cycle
- (max hold time $\leq$ clk-Q + shortest CL)
- Imm generator is always sign-extended to 32 bits
- Control signals
  - PCSel : PC = (PC + 4) or ALU
  - RegWEn : regfile write enable
  - ImmSel : which immediate type to generate (I, S, SB, U, UJ)
  - BrUn : signed or unsigned
  - BrEq : on if rs1 == rs2
  - BrLt : on if rs1 $<$ rs2 based on BrUn
  - BSel : rs2 or immediate
  - ASel : rs1 or PC
  - ALUSel : which ALU operation to perform
  - MemRW : memory read (0) or write (1)
  - WBSel : send PC+4, ALU or memory output to regfile
- Pipelining involves adding registers in b/w operations to create checkpoints
- Latency : time for 1 instruction to finish (longest stage time $\cdot$ stage #)
- Throughput/Bandwidth : how many instructions to complete in a certain amount of time (1 / max stage time)
- Hazards
  - Structural : a required resource is busy (used in multiple stages), e.g. regFile read/write (solve with double pumping)

– Data : data dependency b/w instructions (solve with forwarding after Execution stage; can't solve loads, e.g. lw always requires a stall)

– Control : flow of control dependency, i.e. branches (if taken or unknown) and jumps; branch prediction (if correct) requires no stalls

Instruction Timing

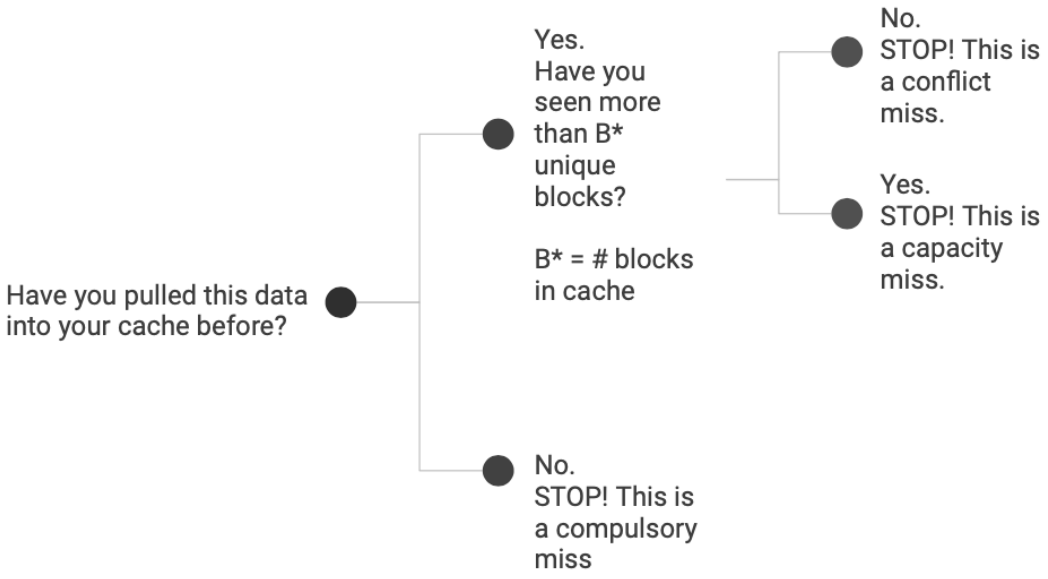
Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X			500ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	*(Op)*	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	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	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

Caches

- Total bits = $\log_2$(bytes in main memory)
- Offset bits = $\log_2$(bytes in a block)
- Index bits = $\log_2$(# of blocks/sets a piece of data can be in)
  FA = 1 set; DM = cache size / block size sets (blocks); N-way SA = cache size / (N * block size) sets
- Tag bits = Total - Offset - Index
- Fully Associative (FA) : max associativity
  - Cache : Valid, Tag, Offest, Replacement Policy
  - 0 index bits (anything can go anywhere)
  - Goes down the cache in order (if new) and replacement policy (LRU take out the 11)
  - Slow check on hit (check every slot), good worst case
  - Worst case still fills cache, i.e. more efficient on avg
- Direct Mapped (DM) : associativity = 1
  - Memory item : Tag, Index, Offset
  - Cache : (I) Valid, T, O
  - Index determines where it goes in the cache, then check valid bit, then check if tag matches
  - Fast check on hit (check 1 slot), bad worst case
  - Worst case may only use 1 slot, used when believing in regularity
- N-Way Set Associative (SA) : associativity in b/w
  - Always tag, index, offset still, but # of index bits could be changed
  - Have a set of blocks (slots) instead of 1 block that each byte of memory could map to, i.e. each set is a N-block FA, and considering only the (cache size / (N * block size))-many sets, it's a DM
  - Double associativity, then decreases index bit (less # of sets)
  - Medium check on hit (all slots in the set specified by the index), medium worst case
  - Worst case : pattern of at least $N + 1$ that maps into same set
- Write-through (no dirty) vs. Write-back (dirty bit)
- Avg memory access time (AMAT) = hit time + miss rate $\times$ miss penalty
- Global hit rate = total hits / total accesses
- Cache Misses (see flow chart)
  - Compulsory : first access of this block
  - Capacity : mostly for FA, fix with a bigger cache
  - Conflict : mostly for DM, fix with higher associativity
  - N.B. Once we've seen B (# of blocks) unique blocks, every miss that's not a compulsory miss is a capacity miss.


Have you pulled this data into your cache before?
Yes.
Have you seen more than B* unique blocks?
B* = # blocks in cache
No.
STOP! This is a conflict miss.
Yes.
STOP! This is a capacity miss.
No.
STOP! This is a compulsory miss

Extra Sanity Checks

- 1 byte = 8 bits; 1 word = 4 bytes = 32 bits
- PC is a register
- ALU is a combinational block
- Consider single CPU cycle starting on (the rising edge of clock on) the PC
- Imm generator is always sign-extended to 32 bits
- N.B. double check int length, uint_32 = 4 bytes regular; uint_8 = 1 byte
- N.B. double check if cache is block aligned
- In C, $2^x = 1 << x$
  bitwise &, |, logic &&, ||
  cond ? true_case : false_case