CS61C MT2 Study Guide - Matthew Tang [Updated 10/25/20 4:28PM]

1 Floating Point

– First 1 is not recorded
– **Sign**: 1 bit
– **Exponent**: 8 bits
– **Significand**: 23 bits (LSB: 10^{-23})
  ∘ Range of $[1.2 * 10^{-38}, 3.4 * 10^{38}]$
  ∘ **Overflow**: Exponent > 8 bits ($> 3.4 * 10^{38}, < -3.4 * 10^{38}$)
  ∘ **Underflow**: Negative exponent > 8 bits
  ∘ Significand is always between 0, 1 (for normalized)
– **Value** $= (-1)^{\text{Sign}} * (1 + \text{Significand}) * 2^{\text{Exponent}-Bias}$
  ∘ Bias $= (2^{\text{expLen}-1} - 1) = 127$
– **Denorm**: No implied leading 1, has exp $-(2^{\text{expLen}}-2) = -126$
  ∘ **Smallest (denorm) number**: $2^{-(\text{bias}-1)} * 2^{-(\text{sigLen})}$
– **Precision**: count of the number bits used to represent a value
– **Accuracy**: Difference between actual value and comp repr
– FP Addition
  ∘ De-normalize to match exponents
  ∘ Add significands
  ∘ Keep same exponent
  ∘ Normalize again
– **Largest (finite) number**: $2^{\text{exp}-\text{bias}} * (2 - 2^{-\text{expLen}})$
– More significand bits = more NaNs, less finite numbers
– Same number of pos numbers < 2 regardless of manLen and expLen
– More accurate long decimal: More significand bits
– **Step size**: $2^{\text{exp}-bias-sigLen}$
  ∘ Number before $2^{\text{exp}-bias}$ was a distance of step size
– **Number of NaN**: $2^{\text{sigLen}+1} - 2$

Exponent	Significant	Object
0	0	0
0	nonzero	Denorm
1-254	anything	± float
255	0	$\pm\infty$
255	nonzero	NaN

2 RISC-V

– **Little-endian**: Least significant byte is smallest address
  ∘ Bits always stored with most significant in upper pos

2.1 Operations

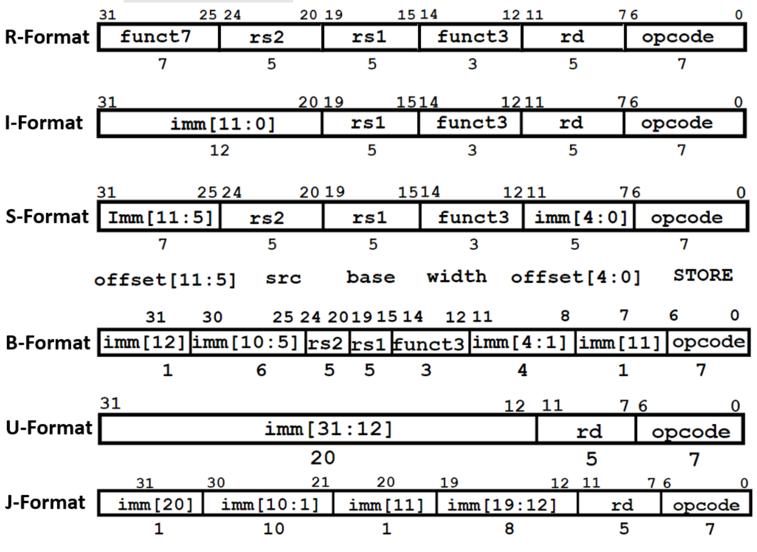
– No logical not, xor with 11111111
– Shift right arithmetic moves n bits to the right, insert sign bit into empty bits
  ∘ Not equivalent to diving by 2^n (fails for odd nums)
– **Program Counter (PC)**: Points to the instruction to be executed
– **Steps to calling functions**
  1) Put arguments in place where func can access them
  2) Transfer control to Function
  3) Acquire local storage resources needed for func
  4) Perform desired task of func
  5) Put return value where code can access it, restore used registers, release local storage
  6) Return control to origin
– Use stack pointer `sp` to point to bottom of stack frame
  ∘ Stack grows downward (addr decreases when we push)
  ∘ Push decrements sp, pop increments sp
  ∘ Stack frame includes return instruction address, args, local vars
– **Memory allocation**: Text, Static, Heap, Stack

2.2 Calling Convention

– Register conventions: a set of rules for which registers are unchanged after `jal`
  ∘ Preserved: `sp, gp, tp, s0-s11` (s0 is fp)
  ∘ Not preserved: `a0-a7, ra, t0-6` (arg and temp registers)
  ∘ Caller must preserve args/rv, temp, ra
  ∘ Callee must preserve saved registers and sp

3 Machine Code

– **R: Register-register arithmetic ops**
  ∘ `funct7` and `funct3` specify the operation
– **I: Register-immediate arithmetic ops**
  ∘ 12 bit immediate covers range $[-2^{11}, 2^{11}-1] = [-2048, 2047]$
  ∘ Sign extend the immediate (for loads too)
  ∘ Loads: load from `rs1` into `rd`
  ∘ Shift amounts limited to 5 bits
– **S: Stores**
  ∘ `sw rs2, imm(rs1)`
– **B(SB): Branches**
  ∘ Used for loops and if
  ∘ Uses PC-relative addressing
  ∘ 12bit immediate always has 13 bits with LSB = 0
  ∘ Range of imm values: $2 * [-2^{n-1}, 2^{n-1} - 1] = [-2^{12}, 2^{12} - 2]$
  ∘ Range of branch instr: $\frac{1}{4}(2)[-2^{n-1}, 2^{n-1}-1] = [-2^{10}, 2^{10}-1]$
  ∘ `beq` with offset 0 is infinite loop
– **U: 20-bit upper immediate instructions**
  ∘ Only `lui` and `auipc`
  ∘ To branch $> 2^{10}$ away, replace `beq` with `bne, j`
  ∘ Use `lui` for upper 20 bits, then `addi` for lower 12
    ∗ `addi` sign extends which can cause problems, need to add 1 to lui value
    ∗ `li` psuedo-op handles this
– **J(UJ): Jumps**
  ∘ `jal` saves PC + 4 to rd
  ∘ **Jump range:** $2 * [-2^{n-1}, 2^{n-1} - 1] = [-2^{20}, 2^{20} - 2]$
  ∘ **Inst Jump range:** $\frac{1}{2}(2)[-2^{n-1}, 2^{n-1} - 1] = [-2^{18}, 2^{18} - 1]$
  ∘ jalr uses I-format (`rd = PC + 4, PC = rs + imm`)
    ∗ Cannot assume LSB is 0, range reduced
    ∗ Use `auipc, jalr` to jump to pc-relative 32 bit offset



4 CALL

4.1 Compiler

– **Input**: High level lang code (`foo.c`)
– **Output**: Assembly code: (`foo.s`)
- Can contain pseudo-instructions

4.2 Assembler

– **Input**: Assembly code (`foo.s`)
– **Output**: Object code (`foo.o`)
– Reads and uses directives
- Ex. `.text` (code), `.data` (static), `.globl sym` , `.string str` `.word w1...wn`

– Replace psuedo-inst, create machine code, create object file
– Need to sign extend immediates
– Branch immediates count halfwords
- PC-relative br/j: count half words between curr and dest

– **2 pass solution**: Solves forward ref problem
- Pass 0: Replace pseudo-instructions
- Pass 1: Remember position of label
- Pass 2: Use label positions to generate code

– **PIC**: Position-independent code
– **Static data refs**:
- `lui` `addi` require full 32b address
- `auipc` `addi` uses PIC

– **Symbol table**
- List of items that can be used by other files
- Labels: function calling
- Data: `.data` variables can be accessed across files

– **Relocation table**
- List of items whose address the file needs
- Absolute labels (`jal, jalr`)
- Static data (`la`)

– **Object file format** (ex. ELF)
- **Object file header**: size, pos of other pieces
- **Text segment**: machine code
- **Data segment**: static data (in binary)
- **Relocation info**: Has lines of code that need to be fixed
- **Symbol table**: Labels/static data that can be referenced
- **Debugging info**

4.3 Linker

– **Input**: Object code files, info tables (`foo.o`)
– **Output**: Executable code (`a.out`)
– Combines several object files into a single executable
– Throws error for duplicate or missing symbols
– **Types of addresses**
- **PC-Relative**: Never need to relocate
  * `beq, bne, jal, auipc/addi`
- **Absolute or external func address**: Always relocate
  * (`auipc/jalr`)
- **Static data reference**: Always relocate
  * `lui/addi`

– **Relocation editing**:
- **J-format**: jal address
- **I,S format**: static pointers
- No conditional branches

– **Statically-linked approach**: include entire library
- Self-contained, includes whole library even if not all is used

– **Dynamically-linked** (DLL): include library at runtime
- Smaller storage, requires less memory
- Requires time to link at runtime (at loader)
- Link at machine code level

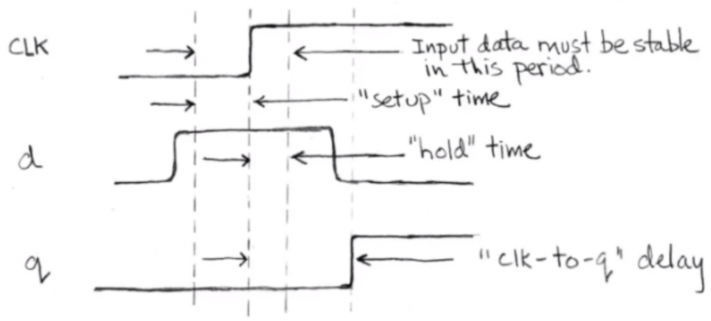
4.4 Loader

– **Input**: Executable code (`a.out`)
– **Output**: program run
– Part of the OS
– Read header to know size of text and data
– Create address space for text, data, stack
– Place static data into memory
– Copy instructions and data from executable file to new address space
– Copy CLI args to stack
– Initialize machine registers (mostly cleared)
– Jump to start-up routine that copies program args from stack to register, set PC

5 Circuits

– **Transistor**: semiconductor device to amplify or switch signals
- Contains Drain, Gate, Source

– **Combinatorial Logic** (CL): Output function of inputs
– **State elements**: Circuits that store info (registers, cache)
– **Flip flop**: Rising edge-triggered sample
- Input d, CLK, Output q
- Setup time: Time before rising edge input must be stable
- Hold time: Time after rising edge input must be stable
- CLK-to-q delay: Time after rising edge until q gets updated and is stable



– **Max delay**: CLK-to-q delay + longest CL + Setup time
- Aka min clock cycle time, critical path delay
- **Critical path**: Longest path from one register to another
- CLK-to-q + longest CL + setup ≤ clk˙cycle
- Max clock rate: 1 / (max delay)

– **Max hold time**: CLK-to-q delay + shortest CL
- hold time ≤ CLK-to-q + shortest path

– **Finite State Machine**: Represent a state transition diagram

5.1 Adder

– LSB: $s_0 = a_0 XOR b_0$, $c_1 = a_0 \& b_0$
– ith bit: $s_i = XOR(a_i, b_i, c_i)$
- $c_{i+1} = MAJ(b_i, b_i, c_i) = a_i b_i + b_i c_i + a_i c_i$

– N 1-bit adders cascaded is a 1 N-bit adder
– For unsigned numbers, existence of carry indicates overflow
– For signed numbers, carry is not indicative
- For highest adder: overflow = $c_n XOR c_{n-1}$
  * No c_{out}, no c_{in}: no overflow
  * c_{in} and c_{out}: no overflow
  * c_{in} but no c_{out}: overflow $(A, B > 0)$
  * c_{out} but not c_{in}: overflow

– **Subtractor**: $A - B = A + (-B)$
- Perform two's complement: XOR to flip bits, add SUB bit
- if SUB is high, then we just add that 1 as first c_{in}

5.2 Boolean Algebra

Name	AND form	OR form
Commutative	$AB = BA$	$A + B = B + A$
Associative	$AB(C) = A(BC)$	$A+(B+C) = (A+B)+C$
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 Morgan's	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}(\overline{B})$

5.3 Canonical Form (Sum of Products)

1) Create truth table
2) Select all rows with a 1 in the output
   - For each row, AND the bits (0 for not)
3) OR all the products(ANDs) together

– Strategies: Add and subtract the same term to factor

6 CPU

6.1 Stages of execution

– **Stage 1: Instruction Fetch (IF)**
  - Send addr to IMEM (Instr MEM), read IMEM at addr

– **Stage 2: Instruction Decode (ID)**
  - Generate control signals from instruction bits, generate immediate, read registers from RegFile

– **Stage 3: Execute (EX) ALU**
  - Perform ALU ops, do branch comparison

– **Stage 4: Memory Access (MEM)**
  - Read/write from DMEM (Data Mem)

– **Stage 5: Write Back to Register (WB)**
  - Write back PC + 4 or the result of ALU op or data from memory to RegFile

– lw uses all 5 stages, sw uses all but WB

– Critical path for lw:
  - $t_{clk-q}+t_{IMEM}+t_{Imm}+t_{ALU}+t_{DMEM}+t_{mux}+t_{setup}$

6.2 Control Signals

– `PCSel`: Choose what (ALU output or PC+4) to write to PC
  - 0: PC+4, 1: ALU (for jumps/branches)

– `ImmSel`: Chooses type of immediate to parse as
  - I, S, SB, UJ, * if no immediate

– `BrUn`: Choose if comparison is unsigned
  - 0: Signed, 1: Unsigned, * if no comparison (or using =)

– `ASel`: Chooses rs1 or PC
  - 0: Reg, 1: PC

– `BSel`: Chooses rs2 or imm:
  - 0: Reg, 1: Immediate

– `ALUSel`: Type of arithmetic operation:
  - Most often add

– `BrEq`: Outputs 1 if equal, 0 otherwise
  - 1/0 if beq, * for others

– `BrLT`: Outputs 1 if less than, 0 otherwise
  - 1/0 if blt, * for others

– `MemRW`: choose to enable writing back to memory (Never *)
  - 0: Read only, 1: write (for sw)

– `RegWEn`: Choose to enable writing back to register
  - 0: Read only, 1: Write back to destination reg (Never *)

– `WBSel`: Choose what to write to register
  - 0: PC + 4 (jal), 1: ALU, 2: MEM (lw)

6.3 Control and Status Registers

– **CSR**: monitor status and peripheries
  - Different from register file (4096 of them)

– `csrrw`: Reads CSR into rd, writes rs1 into CSR

– `csrrs`: Reads CSR into rd, uses rs1 as bitmap to set CSR
  - rs1 bits that are 1 are written to CSR

– `csrrc`: Reads CSR into rd, uses rs1 as bitmap to clear CSR
  - rs1 bits that are 1 are written as 0 to CSR

– For immediate variants, use a 5 bit imm instead of rs1

Instr	rd	rs	ReadCSR	WriteCSR
cssrw	x0	*	no	yes
cssrw	!x0	*	yes	yes
cssrs/c	*	x0	no	yes
cssrs/c	*	!x0	yes	yes

7 Pipelining

7.1 Performance

– Iron law of processor performance:
  - $\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$
  - CPI: Cycles per instruction

– $\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Energy}}{\text{Instruction}}$

– Energy Iron Law: Performance = Power * Energy Efficiency

7.2 Pipelining Hazards

– Put registers in between the 5 stages to pipeline

– Better throughput, worse latency per instruction
  - Clock rate is dramatically higher usually

– **Pipelining hazards**: situations that prevent starting the next instruction in the next clock cycle

7.3 Structural hazard

– 2+ instructions compete for single physical resource

– **Sol1**: Instructions take turns, causing some to stall

– **Sol2**: Add more hardware (can always solve)
  - 2 independent read ports and 1 write port
  - 2 separate memories (cache)

7.4 Data hazard

– Data dependency for read/write
  - **Double pump**: Write in first half, read in the second half
  - **Sol1**: Stall using bubble (`nop`)
    * Compiler will rearrange code that is not dependent
  - **Sol2**: **Forwarding -** grab operand from ALU, not RegFile

– **Load Data Hazard**: Loading and then using it next inst
  - Load available after DMEM, but needed 1 cycle earlier
  - **Sol 1**: **Load delay slot**: Need 1 stall, convert next into a
    * Set regWEn, MemRW, don't update PC to convert to `nop`
  - Cycle after that execute the original command
  - Forward from DMEM output to ALU
  - **Sol2**: Put unrelated instruction into load delay slot

7.5 Control hazard

– Earliest to know branch is at end of execution (ALU)

– **Sol1**: Use 2 stall cycles after a branch

– **Sol2**: If branch/jump taken, convert the 2 next to `nop`

– **Branch prediction**: Bit of branch taken last time

– Moving branch comparator to ID stage would incur data hazard and forwarding could fail

7.6 Superscalar processes

– Multiple execution units (CPI < 1)

– Multiple instructions per clock cycle

– Out of order exeuction to reduce hazards

– Outputs are reordered at commit unit

	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	0 (PC + 4)	*	*	0 (Reg)	0 (Reg)	add	0	1	1 (ALU)
ori	*	*	0	I	*	0 (Reg)	1 (Imm)	or	0	1	1 (ALU)
lw	*	*	0	I	*	0 (Reg)	1 (Imm)	add	0	1	2 (MEM)
sw	*	*	0	S	*	0 (Reg)	1 (Imm)	add	1	0	*
beq	1/0	*	1/0	SB	*	1 (PC)	1 (Imm)	add	0	0	*
jal	*	*	1 (ALU)	UJ	*	1 (PC)	1 (Imm)	add	0	1	0 (PC + 4)
bltu	*	1/0	1/0	SB	1	1 (PC)	1 (Imm)	add	0	0	*

$$t_{\text{clk}} \geq t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{RF read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{DMEM read}} + t_{\text{mux}} + t_{\text{RF setup}}$$

$$\begin{aligned}
\textbf{IF}: &\ t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{Reg setup}} = 30 + 250 + 20 = 300 \text{ ps} \\
\textbf{ID}: &\ t_{\text{Reg clk-to-q}} + t_{\text{RF read}} + t_{\text{Reg setup}} = 30 + 150 + 20 = 200 \text{ ps} \\
\textbf{EX}: &\ t_{\text{Reg clk-to-q}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{Reg setup}} + t_{\text{mux}} = 30 + 25 + 200 + 20 + 25 = 300 \text{ ps} \\
\textbf{MEM}: &\ t_{\text{Reg clk-to-q}} + t_{\text{DMEM read}} + t_{\text{Reg setup}} = 30 + 250 + 20 = 300 \text{ ps} \\
\textbf{WB}: &\ t_{\text{Reg clk-to-q}} + t_{\text{mux}} + t_{\text{RF setup}} = 30 + 25 + 20 = 75 \text{ ps}
\end{aligned}$$

RISC-V Instruction Set

Core Instruction Formats

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

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`add`	ADD	R	`0110011`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	
`xor`	XOR	R	`0110011`	`0x4`	`0x00`	`rd = rs1 ˆ rs2`	
`or`	OR	R	`0110011`	`0x6`	`0x00`	`rd = rs1 | rs2`	
`and`	AND	R	`0110011`	`0x7`	`0x00`	`rd = rs1 & rs2`	
`sll`	Shift Left Logical	R	`0110011`	`0x1`	`0x00`	`rd = rs1 << rs2`	
`srl`	Shift Right Logical	R	`0110011`	`0x5`	`0x00`	`rd = rs1 >> rs2`	
`sra`	Shift Right Arith*	R	`0110011`	`0x5`	`0x20`	`rd = rs1 >> rs2`	msb-extends
`slt`	Set Less Than	R	`0110011`	`0x2`	`0x00`	`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`	`0x00`	`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`		`rd = rs1 ˆ imm`	
`ori`	OR Immediate	I	`0010011`	`0x6`		`rd = rs1 | imm`	
`andi`	AND Immediate	I	`0010011`	`0x7`		`rd = rs1 & imm`	
`slli`	Shift Left Logical Imm	I	`0010011`	`0x1`	`imm[5:11]=0x00`	`rd = rs1 << imm[0:4]`	
`srli`	Shift Right Logical Imm	I	`0010011`	`0x5`	`imm[5:11]=0x00`	`rd = rs1 >> imm[0:4]`	
`srai`	Shift Right Arith Imm	I	`0010011`	`0x5`	`imm[5:11]=0x20`	`rd = rs1 >> imm[0:4]`	msb-extends
`slti`	Set Less Than Imm	I	`0010011`	`0x2`		`rd = (rs1 < imm)?1:0`	
`sltiu`	Set Less Than Imm (U)	I	`0010011`	`0x3`		`rd = (rs1 < imm)?1:0`	zero-extends
`lb`	Load Byte	I	`0000011`	`0x0`		`rd = M[rs1+imm][0:7]`	
`lh`	Load Half	I	`0000011`	`0x1`		`rd = M[rs1+imm][0:15]`	
`lw`	Load Word	I	`0000011`	`0x2`		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	`0000011`	`0x4`		`rd = M[rs1+imm][0:7]`	zero-extends
`lhu`	Load Half (U)	I	`0000011`	`0x5`		`rd = M[rs1+imm][0:15]`	zero-extends
`sb`	Store Byte	S	`0100011`	`0x0`		`M[rs1+imm][0:7] = rs2[0:7]`	
`sh`	Store Half	S	`0100011`	`0x1`		`M[rs1+imm][0:15] = rs2[0:15]`	
`sw`	Store Word	S	`0100011`	`0x2`		`M[rs1+imm][0:31] = rs2[0:31]`	
`beq`	Branch ==	B	`1100011`	`0x0`		`if(rs1 == rs2) PC += imm`	
`bne`	Branch !=	B	`1100011`	`0x1`		`if(rs1 != rs2) PC += imm`	
`blt`	Branch <	B	`1100011`	`0x4`		`if(rs1 <  rs2) PC += imm`	
`bge`	Branch $\leq$	B	`1100011`	`0x5`		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	`1100011`	`0x6`		`if(rs1 <  rs2) PC += imm`	zero-extends
`bgeu`	Branch $\geq$ (U)	B	`1100011`	`0x7`		`if(rs1 >= rs2) PC += imm`	zero-extends
`jal`	Jump And Link	J	`1101111`			`rd = PC+4; PC += imm`	
`jalr`	Jump And Link Reg	I	`1100111`	`0x0`		`rd = PC+4; PC = rs1 + imm`	
`lui`	Load Upper Imm	U	`0110111`			`rd = imm << 12`	
`auipc`	Add Upper Imm to PC	U	`0010111`			`rd = PC + (imm << 12)`	
`ecall`	Environment Call	I	`1110011`	`0x0`	`imm=0x0`	`Transfer control to OS`	
`ebreak`	Environment Break	I	`1110011`	`0x0`	`imm=0x1`	`Transfer control to debugger`	

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning

`la rd, symbol`	`auipc rd, symbol[31:12]`
	`addi rd, rd, symbol[11:0]`
		Load address

`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]`
	`l{b|h|w|d} rd, symbol[11:0](rd)`
		Load global

`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`s{b|h|w|d} rd, symbol[11:0](rt)`
		Store global

`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{w|d} rd, symbol[11:0](rt)`
		Floating-point load global

`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{w|d} rd, symbol[11:0](rt)`
		Floating-point store global

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine

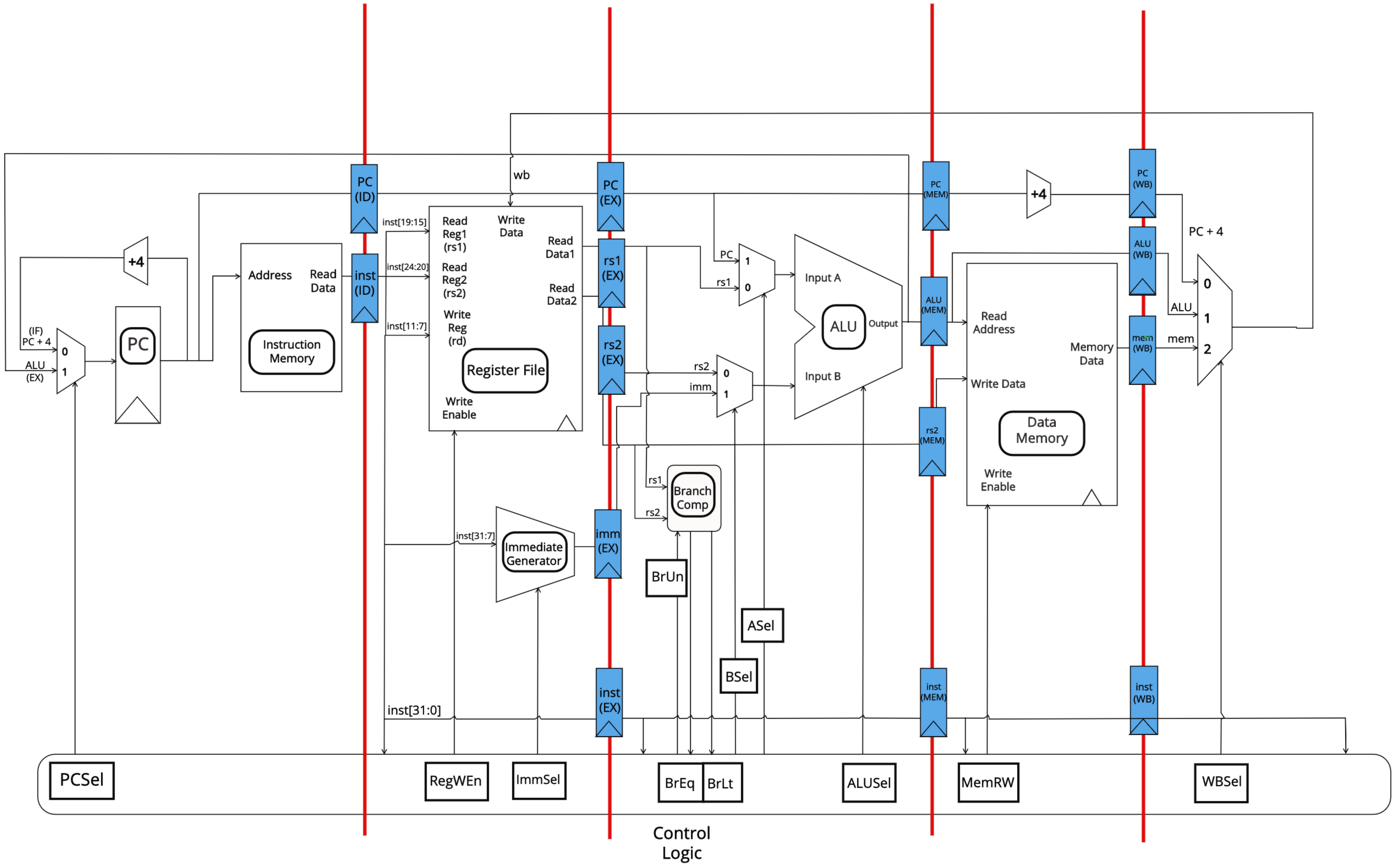
Registers

Register	ABI Name	Description	Saver
`x0`	`zero`	Zero constant	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	—
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	Callee
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0 / fp`	Saved / frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Fn args/return values	Caller
`x12-x17`	`a2-a7`	Fn args	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP args/return values	Caller
`f12-17`	`fa2-7`	FP args	Caller
`f18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

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

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK



(IF) PC + 4
ALU (EX)
0
1
+4
PC
Address
Read Data
Instruction Memory
PC (ID)
inst (ID)
inst[19:15]
inst[24:20]
inst[11:7]
Read Reg1 (rs1)
Read Reg2 (rs2)
Write Reg (rd)
Write Data
wb
Read Data1
Read Data2
Register File
Write Enable
inst[31:7]
Immediate Generator
inst[31:0]
PC (EX)
rs1 (EX)
rs2 (EX)
imm (EX)
inst (EX)
PC
rs1
rs2
imm
Input A
Input B
ALU
Output
rs1
rs2
Branch Comp
BrUn
ASel
BSel
PC (MEM)
ALU (MEM)
rs2 (MEM)
inst (MEM)
+4
Read Address
Write Data
Memory Data
Data Memory
Write Enable
PC (WB)
ALU (WB)
mem (WB)
inst (WB)
PC + 4
ALU
mem
2
PCSel
RegWEn
ImmSel
BrEq
BrLt
ALUSel
MemRW
WBSel
Control Logic