# COM SCI M151B Week 1

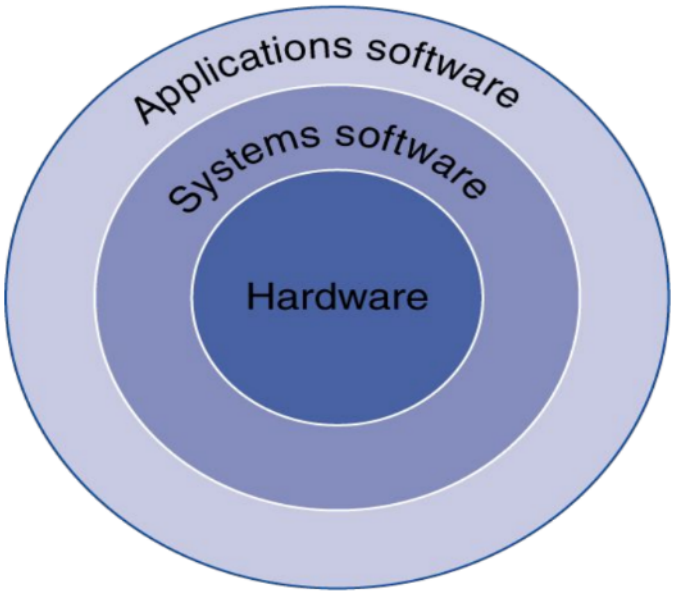
Aidan Jan

October 3, 2024

## Using Abstraction

- We see a computer as a *box* with multiple layers of **abstraction**.
- Depending on which layer we want ot work on, we *abstract away* the irrelevant layers.
  - The *main benefit* is that we don't need to know the unnecessary details of the other layers in order to be able to work on our layer.

### Computer Abstractions (simplified)




- Application software
  - Translation from algorithm to code
  - Written in high-level language (e.g., C, Java)
- System software
  - Compiler: translates high level language code to machine code
  - Operating system: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks and sharing resources

- Hardware
  - Processor memory
  - I/O controllers

`https://www.youtube.com/watch?v=_y-5nZAbgt4`

## How do computers work?

- Theoretical and Historical Points of View
  - Turing machines and history of electronics and computers.

## Level of Program Code

- High level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
  - Architecture-dependent
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

## How to maintain compatibility?

There are many different types of computer architecture, and many different languages applications are written in. How do we maintain compatibility?

- System software (OS) and software makes a contract to always give a program with **only a set of known instructions**, and the hardware promises to be able to run that.
- The **ISA** is the *interface* between hardware and software. This allows the HW and SW to change/evolve **independently**.
- Software sees:
  - Function description of hardware:
    1. Storage locations (e.g., memory)
    2. Operations (e.g., add)
- Hardware sees:
  - List of instructions and their order
- In this course, we will use RISC-V ISA.

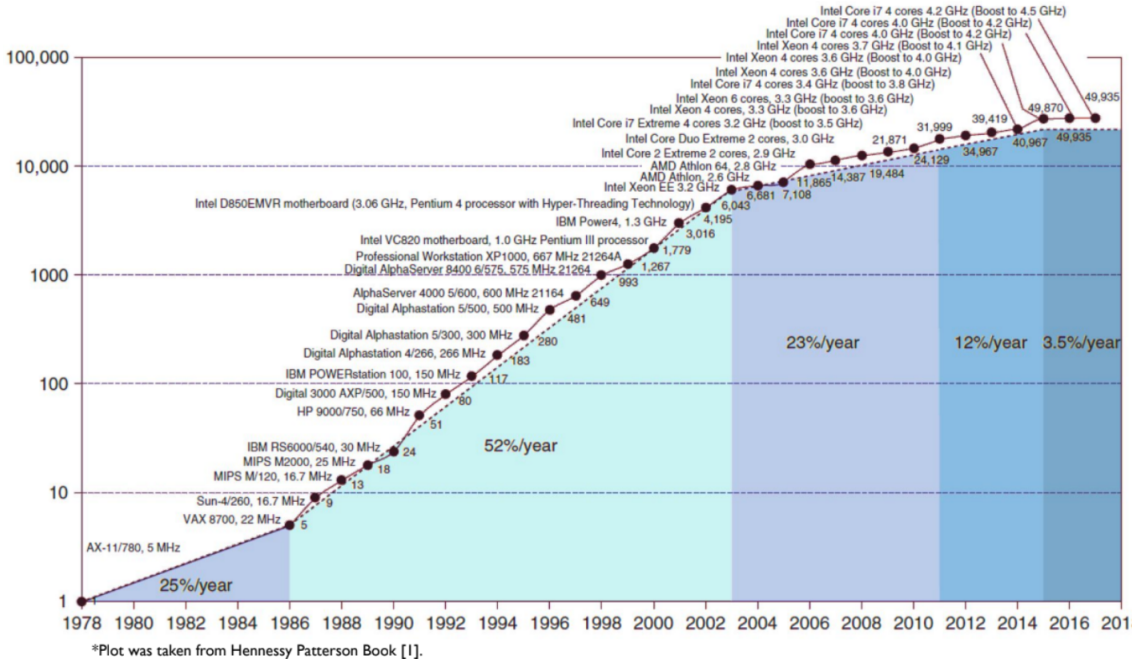
## Goals when designing computers

They must be efficient. What is *efficient*?

- Performance (often most important)
- Power consumption
- Cost
- Reliable and Secure

### Past, Present, and Future

- Moore's Law:
  - We started with the ENIAC after World War II, and that computer had a size of 1000 cubic feet, used 125kW of power, only does 2000 additions per second, and had 48 kB of memory.
  - Now, we have computers taking up 1 cubic foot, consumes 250W of power, capable of 20B operations per second, with multiple GB of memory, for only less than 0.01% of the cost of the ENIAC.
  - Moore's Law states that every two years, the number of transistors (and therefore computational power) doubles. This suggests that the cost per transistor reduces each year.



*Plot was taken from Hennessy Patterson Book [1].

  - From 1986 to 2003, Moore's Law was theorized
  - At 2003 to 2011, transistors grew so close together that it became near-impossible to make them smaller; refer to Power Wall below.
  - From 2011 to 2015, Amdahl's Law started taking effect. (Refer below).
  -

- Dennard's Scaling Law:
  - According to Moore's law, the number of transistors on a chip doubles every two years, thus each transistor's area is reduced by 50%, or every dimension by 0.7x.

  - As a result, voltage is reduced by -30% (0.7x) to keep the electric field constant. $V = EL$
  - $L$ is reduced, thus delays are reduced by -30%. ($x = Vt$)
  - Frequency is increased by +40% ($f = 1/t$)
  - Capacitance is reduced by -30% ($C = kA/L$)
- Scaling Power and Energy

$$P = CV^2 f$$

  - Power consumption per transistor is decreased by -50%.
  - Power consumption of the entire chip stays the same! Except now it has more transistors.
- Where are these improvements coming from?
  1. Advancements in microelectronics and fabrication technologies.
  2. Advancements in architectural techniques
     - What this course is about!
     - This lead to an improvement by a factor of 25 versus if we had only relied on (1.)
- Power Wall
  - Up to 2005, manufacturers could double the processor performance while keeping the power constant.
  - Since 2005, due to smaller transistor sizes, **static power leakage** becomes so dominant that the power consumption did not stay constant. (Moore's Law slowed down.)
  - However, the rate still grew since multicore systems began to emerge.
- Amdahl's Law
  - Performance improvement (speedup) is limited by the part you cannot improve (called "sequential part").

$$\text{speed up} = \frac{1}{(1-p) + \frac{p}{s}}$$

    * $P$ is the part that can be improved (e.g., **parallelized**)
    * $S$ is the factor for improvement (e.g., more cores for parallelization)
- Dark Silicon
  - Where we are right now
  - Due to thermal management limitations, some parts of a chip cannot be turned on.
  - This is the end of the multi-core era.
  - New technologies and techniques are being used to continue the scaling. We need cross-stack approaches!

# Instruction Set Architecture (ISA)

## What is an ISA

- The contract between software and hardware is the ISA. Typically described by giving all the programmer-visible state (register + memory) plus the semantics of the instructions that operate on that state.
- IBM 360 was the first line of machines to separate ISA from implementation (aka. *microarchitecture*)
- There are **many implementations possible** for a given ISA

  - AMD, Intel, VIA processors run the AMD64 ISA
  - Many cell phones use the ARM ISA with implementations from many different ompanies including Apple, Qualcomm, Samsung, Huawei, etc.
- We use RISC-V as the standard ISA in class.

## Design Methodology

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator → hardwired, unpipelined
  - CISC → microcoded
  - RISC → hardwired, pipelined
  - VLIW → fixed-latency in-order parallel pipelines
  - JVM → software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor

## RISC-V

### What is RISC-V?

- An *open-source* "RISC"-based ISA (*royalty-free*)
- Developed in the 2010s at Berkeley
- Mostly maintained by the open-source community.
- Different extensions and models. We will focus on the 32-bit "base" mode (i.e., "RV321").
- RISC = "Reduced Instruction Set Computers". Unlike "CISC", every instruction in RISC is the same size.

### RISC vs. CISC

- Instruction size of RISC is fixed, CISC is variable
- RISC has simple (one-by-one) operations, rather than packed operations
- RISC is less complex than CISC in terms of instruction set.

## Widely-Used ISAs

- All "RISC" except x86 (Intel's ISA)
- Most popular RISC ISAs:
  - MIPS, ARM, PowerPC, and RISC-V

## Stored Program Computer (von Neumann)

- Computer hardware is a machine that reads instructions one-by-one and executes them sequentially. It continues this until the program finishes.
- Memory holds *both* program and data
  - Instructions and data in a linear memory array
  - Instructions can be modified as data
- Sequential instruction processing
  1. Program Counter (PC) identifies current instruction.
  2. Fetch instruction from memory.
  3. Update state (e.g., PC and memory) as a function of current state according to instruction
  4. Repeat.

## How to build an ISA?

- Transition from HLL to assmebly
- Take CS132 (Compiler Construction)

## What do instructions look like?

There are two main parts: commands and operands.

- In 32-bit RISC-V, each instruction is fixed-size and is 32-bit.
- Possible Types of Operands:
  1. Registers
     - A small storage unit *inside* the processor to quickly access data.
     - Faster than memory, but much smaller (smaller is faster!)
     - Can be seen by Software*! (This is something that we will clarify later.)
     - Typically between 16 and 64 registers in modern ISAs.
       * Larger width means easier data transfer but more power.
       * More registers means less access to the main memory, but requires more area and more power.
       * Low-end processors use smaller registers. High-Performance processors use wider and more registers.
     - See section on RISC-V Registers
  2. Immediate
     - Constant numbers to be used in an instruction
     - Example: `addi x2, x1, 5`
     - Registers are 32-bits but immediates may not. (This is because they must fit into the instruction. Use pointer instead if it doesn't fit!)
     - See section on RISC-V Immediates.
  3. Memory
     - Values are stored in a memory and could be accessed.
     - Memory shouldbe byte-addressable
     - See section on Memory.

**RISC-V Registers**

- RISC-V (RV32) uses 32 registers, 32-bit each (called "word").
- RV64 uses 32 registers, 64-bit each (called double-word)
- Each register shown as $xi$.
- $x0$ is hardwired to zero.
- Registers are stored in a data structure called the **Register File** (this is a hardware unit that will be discussed later.)
- For more high-end processors, there is a separate set of registers for floating point operations.

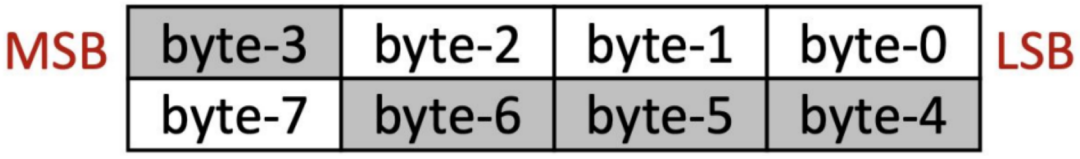
**RISC-V Immediates**

- RISC-V does Sign-Extension and Padding
- Sign-Extension
  - Two's Complement for signed values
- Padding
  - For LSBs (least significant bit), padding zeros is sufficient.

**Memory**

Format and addressing

- Byte-addressability vs. 32-bit data (little endian vs. big endian)
- Alignment:




- Load and Store Variants
  - LW, LB, LH (load different amounts of bits - word, byte, high) (this one does sign extension, e.g., adding 1's instead of 0's if number is negative)
  - LBU, LHU (byte unsigned, high unsigned) (this one does not do sign extension, which means higher bits are unaltered)
  - SW, SB, SH (store word, byte, high - doesn't care about the sign)

## Instruction Types

1. Arithmetic/ALU
   - Do an arithmetic operation on two registers or one register and an immediate
   - Result is saved in another register
   - Example: `ADDI x5, x1, 10` - adds `x1` and `10`, stores in `x5`
2. Memory

- Load and Store
- Addressing modes:
  - Base (register) + Offset (immediate)
  - Base could be zero or it could be a value stored in a register.
- Data Size and format
  - Word, half-word, byte
  - Signed and unsigned

3. Control-Flow
   - Program counter and PC register
   - Next PC $\rightarrow$ PC + 4 (remember that instructions are 4 bytes and memory is byte-addressable).
   - Sometimes, we need to jump/branch over some instructions (GOTO statements!)
     - Conditionals
     - Loops

### RISC-V Arithmetic operators

| | | | |
|---|---|---|---|
| ADDI | `addi rd, rs1, constant` | Add Immediate | `reg[rd] <= reg[rs1] + constant` |
| SLTI | `slti rd, rs1, constant` | Compare < Immediate (Signed) | `reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0` |
| SLTIU | `sltiu rd, rs1, constant` | Compare < Immediate (Unsigned) | `reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0` |
| XORI | `xori rd, rs1, constant` | Xor Immediate | `reg[rd] <= reg[rs1] ^ constant` |
| ORI | `ori rd, rs1, constant` | Or Immediate | `reg[rd] <= reg[rs1] \| constant` |
| ANDI | `andi rd, rs1, constant` | And Immediate | `reg[rd] <= reg[rs1] & constant` |
| SLLI | `slli rd, rs1, constant` | Shift Left Logical Immediate | `reg[rd] <= reg[rs1] « constant` |
| SRLI | `srli rd, rs1, constant` | Shift Right Logical Immediate | `reg[rd] <= reg[rs1] »$_u$ constant` |
| SRAI | `srai rd, rs1, constant` | Shift Right Arithmetic Immediate | `reg[rd] <= reg[rs1] »$_s$ constant` |
| ADD | `add rd, rs1, rs2` | Add | `reg[rd] <= reg[rs1] + reg[rs2]` |
| SUB | `sub rd, rs1, rs2` | Subtract | `reg[rd] <= reg[rs1] - reg[rs2]` |
| SLL | `sll rd, rs1, rs2` | Shift Left Logical | `reg[rd] <= reg[rs1] « reg[rs2]` |
| SLT | `slt rd, rs1, rs2` | Compare < (Signed) | `reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0` |
| SLTU | `sltu rd, rs1, rs2` | Compare < (Unsigned) | `reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0` |
| XOR | `xor rd, rs1, rs2` | Xor | `reg[rd] <= reg[rs1] ^ reg[rs2]` |
| SRL | `srl rd, rs1, rs2` | Shift Right Logical | `reg[rd] <= reg[rs1] »$_u$ reg[rs2]` |
| SRA | `sra rd, rs1, rs2` | Shift Right Arithmetic | `reg[rd] <= reg[rs1] »$_s$ reg[rs2]` |
| OR | `or rd, rs1, rs2` | Or | `reg[rd] <= reg[rs1] \| reg[rs2]` |
| AND | `and rd, rs1, rs2` | And | `reg[rd] <= reg[rs1] & reg[rs2]` |

### Memory-Type Instructions

| | | | |
|---|---|---|---|
| LB | `lb rd, offset(rs1)` | Load Byte | `reg[rd] <= signExtend(mem[addr])` |
| LH | `lh rd, offset(rs1)` | Load Half Word | `reg[rd] <= signExtend(mem[addr + 1: addr])` |
| LW | `lw rd, offset(rs1)` | Load Word | `reg[rd] <= mem[addr + 3: addr]` |
| LBU | `lbu rd, offset(rs1)` | Load Byte (Unsigned) | `reg[rd] <= zeroExtend(mem[addr])` |
| LHU | `lhu rd, offset(rs1)` | Load Half Word (Unsigned) | `reg[rd] <= zeroExtend(mem[addr + 1: addr])` |
| SB | `sb rs2, offset(rs1)` | Store Byte | `mem[addr] <= reg[rs2][7:0]` |
| SH | `sh rs2, offset(rs1)` | Store Half Word | `mem[addr + 1: addr] <= reg[rs2][15:0]` |
| SW | `sw rs2, offset(rs1)` | Store Word | `mem[addr + 3: addr] <= reg[rs2]` |

## Jumps

- Checking a condition (if/else, switch)
- Loops (for/while)
- Function calls and returns
- Exception and interrupt

To jump, a destination address (i.e., where to jump?) is needed. We use *PC-relative addressing*, or we specify the address relative to the PC.

### JUMP vs. Branch

- Jump **always** change the PC!
- Branch changes the PC **only if** the condition is TRUE!

### JUMP and Link

- Why do we need linking?
  - Linking jumps while storing the address you were at before. This is useful for example, function calls, where you have to jump back after reaching the `ret`
- What if we don't need to link?
  - Use `jalr` instead of `jal`
  - `jalr` calculates the destination with a register and offset.
    - `jalr x10 x1 10` would move the execution to the address stored in x1, plus 10 bytes, storing the address of x10
    - `jal x10 label` would move the execution to whereever the label is, and store the address of x10.

### Control Flow Instructions

| | | | |
|---|---|---|---|
| JAL | `jal rd, label` | Jump and Link | `reg[rd] <= pc + 4`<br>`pc <= label` |
| JALR | `jalr rd, offset(rs1)` | Jump and Link Register | `reg[rd] <= pc + 4`<br>`pc <= {(reg[rs1] + offset)[31:1], 1'b0}` |
| BEQ | `beq rs1, rs2, label` | Branch if $=$ | `pc <= (reg[rs1] == reg[rs2]) ? label`<br>`: pc + 4` |
| BNE | `bne rs1, rs2, label` | Branch if $\neq$ | `pc <= (reg[rs1] != reg[rs2]) ? label`<br>`: pc + 4` |
| BLT | `blt rs1, rs2, label` | Branch if $<$ (Signed) | `pc <= (reg[rs1] <`$_s$` reg[rs2]) ? label`<br>`: pc + 4` |
| BGE | `bge rs1, rs2, label` | Branch if $\geq$ (Signed) | `pc <= (reg[rs1] >=`$_s$` reg[rs2]) ? label`<br>`: pc + 4` |
| BLTU | `bltu rs1, rs2, label` | Branch if $<$ (Unsigned) | `pc <= (reg[rs1] <`$_u$` reg[rs2]) ? label`<br>`: pc + 4` |
| BGEU | `bgeu rs1, rs2, label` | Branch if $\geq$ (Unsigned) | `pc <= (reg[rs1] >=`$_u$` reg[rs2]) ? label`<br>`: pc + 4` |

## Pseudo Instructions

- Instructions that are not in the ISA but can be easily converted to one or two.
- Example: (load immediate) `li rd, constant` $\rightarrow$ `addi rd, x0, constant`

| Pseudoinstruction | Description | Execution |
|---|---|---|
| `li rd, constant` | Load Immediate | `reg[rd] <= constant` |
| `mv rd, rs1` | Move | `reg[rd] <= reg[rs1] + 0` |
| `not rd, rs1` | Logical Not | `reg[rd] <= reg[rs1] ^ -1` |
| `neg rd, rs1` | Arithmetic Negation | `reg[rd] <= 0 - reg[rs1]` |
| `j label` | Jump | `pc <= label` |
| `jal label`<br>`call label` | Jump and Link (with `ra`) | `reg[ra] <= pc + 4`<br>`pc <= label` |
| `jr rs` | Jump Register | `pc <= reg[rs1] & ~1` |
| `jalr rs` | Jump and Link Register (with `ra`) | `reg[ra] <= pc + 4`<br>`pc <= reg[rs1] & ~1` |
| `ret` | Return from Subroutine | `pc <= reg[ra]` |
| `bgt rs1, rs2, label` | Branch $>$ (Signed) | `pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4` |
| `ble rs1, rs2, label` | Branch $\leq$ (Signed) | `pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4` |
| `bgtu rs1, rs2, label` | Branch $>$ (Unsigned) | `pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4` |
| `bleu rs1, rs2, label` | Branch $\leq$ (Unsigned) | `pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4` |
| `beqz rs1, label` | Branch $= 0$ | `pc <= (reg[rs1] == 0) ? label : pc + 4` |
| `bnez rs1, label` | Branch $\neq 0$ | `pc <= (reg[rs1] != 0) ? label : pc + 4` |
| `bltz rs1, label` | Branch $< 0$ (Signed) | `pc <= (reg[rs1] <s 0) ? label : pc + 4` |
| `bgez rs1, label` | Branch $\geq 0$ (Signed) | `pc <= (reg[rs1] >=s 0) ? label : pc + 4` |
| `bgtz rs1, label` | Branch $> 0$ (Signed) | `pc <= (reg[rs1] >s 0) ? label : pc + 4` |
| `blez rs1, label` | Branch $\leq 0$ (Signed) | `pc <= (reg[rs1] <=s 0) ? label : pc + 4` |

- `RET` and `CALL` (return and function call) are also pseudo instructions, which are built from jump and load commands

## Calling Convention

- When starting a new mechanism, a set of rules have to be followed to guarantee correctness.
- Major rules:
  - The callee promises to leave some registers unchanged for the caller. If needs to be modified, the callee saves these on the stack first and restores them at the end.
  - On the call, the return has to be saved on the stack.
  - Before returning, the frame pointer has to be recovered.

### Stack Management

- **Caller**
  - Puts arguments on the stack
  - Invokes callee by using call instruction
- **Callee**
  - Saves reserved registers for caller
  - Saves old based pointer
  - Makes room for local variables and executes the code
  - Puts return value into register
  - Restores stack frame and key registers
  - Returns

| Registers | Symbolic names | Description | Saver |
|---|---|---|---|
| x0 | zero | Hardwired zero | — |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | — |
| x4 | tp | Thread pointer | — |
| x5-x7 | t0-t2 | Temporary registers | Caller |
| x8-x9 | s0-s1 | Saved registers | Callee |
| x10-x11 | a0-a1 | Function arguments and return values | Caller |
| x12-x17 | a2-a7 | Function arguments | Caller |
| x18-x27 | s2-s11 | Saved registers | Callee |
| x28-x31 | t3-t6 | Temporary registers | Caller |

## Assembly to Machine Code

- Machine don't understand assembly, so computers use a specific tool called an *assembler* to generate machine code (binary)
- To generate the machine code, the assembler uses a table.

| 31 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 (MIT 6.004 subset)**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[31:12] | | | | rd | 0110111 | LUI |
| 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 | 010 | rd | 0000011 | LW |
| 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 |

- Note that every function uses a different number of bits to store immediates!
- R-type instructions have funct3 and funct7 (instruction bits are separated) to preserve modularity.
- Notice that each group of functions (ALU, Memory, Control) share the same opcode. The assembler distinguishes them based on the funct3 and funct7.

**How to build an ISA?**

- Translition from HLL to assembly
- No discussion (yet) on how
  1. to generate machine code?
  2. to upload this to hardware?
  3. does hardware run this?

**How to execute these codes?**

- Store-Program computers!
- To run programs on the processor, we store programs on the memory (same as data).
- Processor reads one instruction, updates some states and/or some memory contents, then reads the next instruction, ...
- This model called Store-Program (or von-Neuman model). (there are other alternatives too!)