RISC-V assembly

Quick review: data as bits

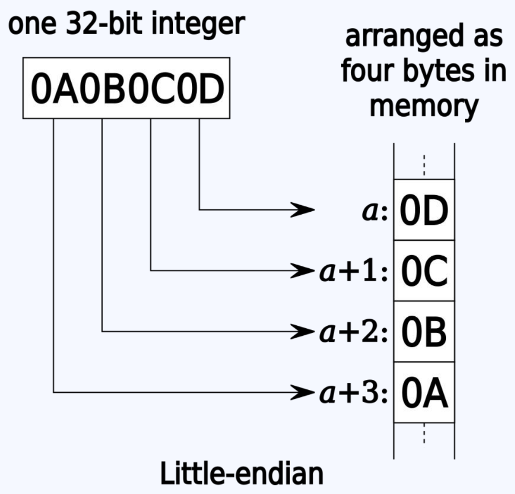
Same bits in memory are interpreted differently based on context

Each byte in memory has its own address

Code demo on signed/unsigned numbers

Keep in mind:

- Signed numbers are negated using 2s complement
- A signed negative number will always have a 1 as the MSB
- Cast to larger numbers: sign-extend (copy MSB) for signed, zero-pad for unsigned



By Aeroid - Own work, CC BY-SA 4.0, link

? ? ?

>> has different behavior on ints vs uints
do you think the decision to use logical vs. arithmetic
shift is made by the compiler or the CPU?

Aligned addressing

Each memory address refers to a byte

32-bit integers *typically* have addresses that are multiples of 4 (why?)

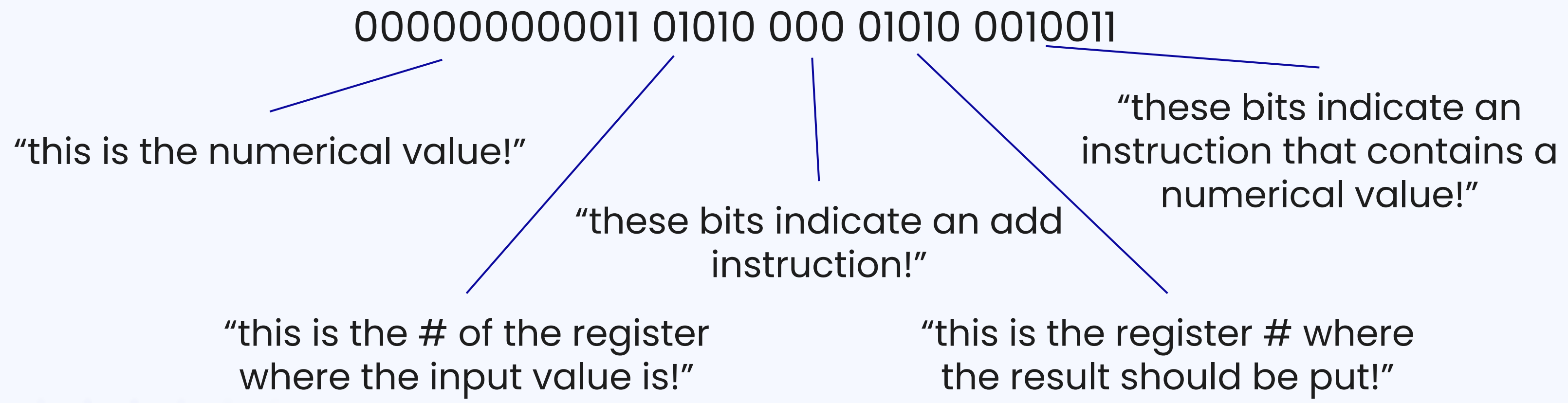
Mostly up to the compiler; constrained by ISA

Address	Value
0x0000	0x4d
0x0001	0xf0
0x0002	0x00
0x0003	0x18

Preview: interpreting instructions

We could interpret ‘0xe7` as 233, -23, or ← based on context

Similarly, we can interpret `0x00350513’ as 3474707 or the instruction “increment register 10 by 3”



000000000011 01010 000 01010 0010011

“this is the numerical value!”

“this is the # of the register where the input value is!”

“these bits indicate an add instruction!”

“this is the register # where the result should be put!”

“these bits indicate an instruction that contains a numerical value!”

Intermission: why are you taking this course?

- Learn what different terms/ideas mean (pipelining, branch prediction, how a CPU can take different time on different instructions)
- Understand computer specs when buying parts
- Pull back the magic curtain
- Impress friends and family with computer knowledge
- Understand how hardware affects SW performance
- Hows and whys of specialized hardware
- Understand low-level systems/academic papers about low-level systems

? ? ?

What sorts of operations do we want a computer to be able to do?

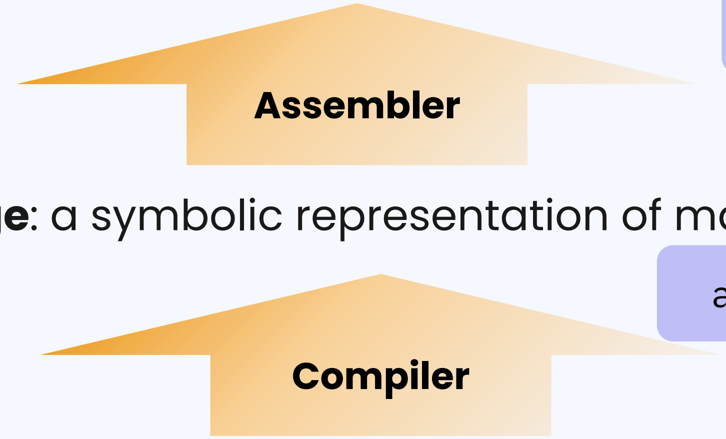
Definitions (from P&H Chapter 2)

Instruction: a command computer hardware understands and obeys

part of ISA

Machine language: a binary representation of machine instructions

```
0x00150513
```



Assembly language: a symbolic representation of machine instructions

```
addi a0, a0, 1
```

High-level language: a portable language such as C, C++, Java that is composed of words and algebraic notation

```
x++;
```

Why RISC-V?

We'll be using RISC-V in class and for the next few assignments

Stems from research out of Berkeley in the 80s

Completely open

Base instruction set is **simple but makes a working computer**

Started as academic, starting to see real-world use

Important note: not all ISAs are like RISC-V. We'll keep highlighting the similarities/differences between ISAs throughout the course

Three basic classes of instructions in most ISAs

Computations

Add two variables together, subtract a constant, perform bit operations...

Data transfer

Loads and stores from memory

Control logic

Conditional and unconditional jumps (support for if-statements, loops, function calls)

Data needs to be easily accessible by the CPU in order to do this

Registers

Small, fast memory (usually the size of a word, or default data size of a specific architecture)

Used by the CPU

Usually addressed separately from main memory

Some have special purposes, some are general purpose (GPR)

In RISC-V, **all arithmetic and control computations are done on registers**

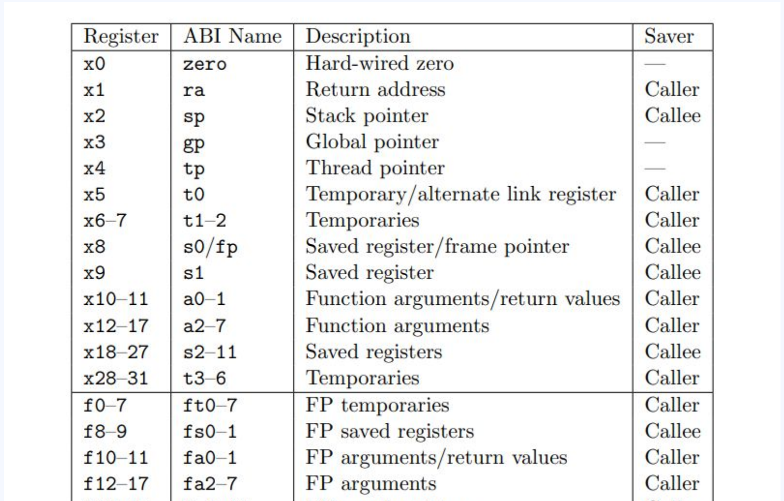
- To compute on memory, first need to load data from memory into register
- Called a “register-register” or “load-store” architecture (other example: arm)
- Contrast with “register-memory” architecture (example: x86)

RISC-V Specification

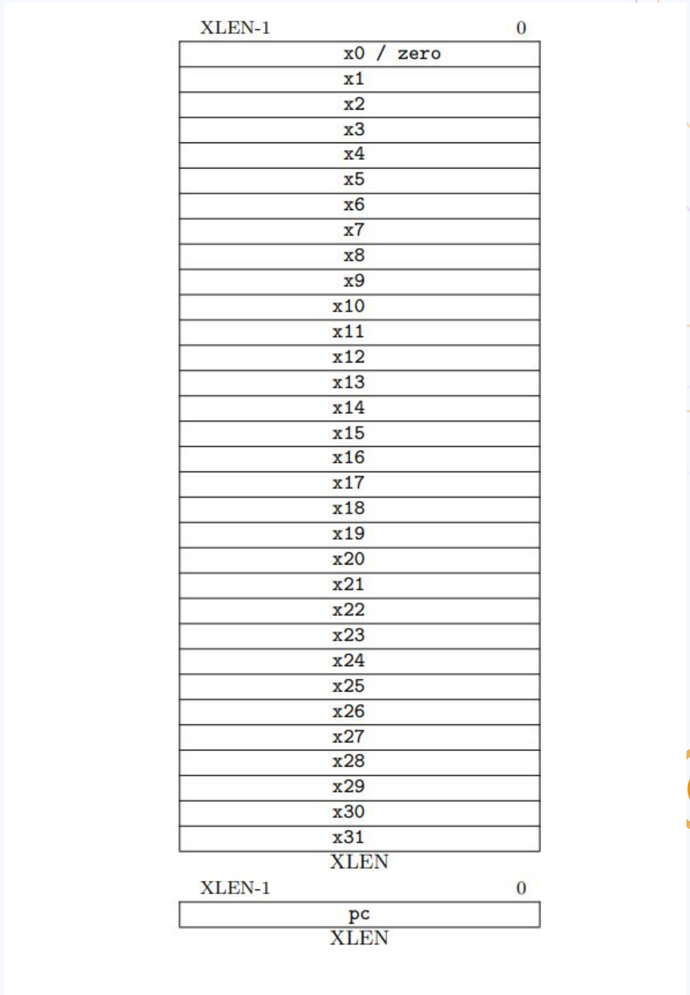
RV32I: 31 GPRs (x1-x32) – conventions define the role of some of these

x0 hard-wired to 0

pc (program counter) – holds address of current instruction



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller



? ? ?

ISA: model of the computations a CPU can do (interface for programmer/compiler)
Microarchitecture: hardware implementation of ISA (multiple microarchitectures possible for one ISA, e.g. different Intel chips implementing Intel64)

If the ISA is about instructions and microarchitecture is about hardware, why does the ISA spec define how many registers are available?

Arithmetic and logical operations

RISC-V (32I base instruction set) offers the following:

Arithmetic: ADD, SUB

Logical: AND, OR, XOR

Shift: SLL, SRL, SRA

Comparison: SLT[U]

No need for ADDU because 2s complement math "just works" (see textbook)
Compiler takes care of applying SRL to unsigned vars and SRA to signed

Three operands: two source registers (*rs1*, rs2), one destination (**rd**)

```
sub x12, x11, x10 # x12 = x11 - x10
or x13, x13, x14 # x13 |= x14
slt x17, x15, x16 # x17 = x15 < x16 ? 1 : 0
```

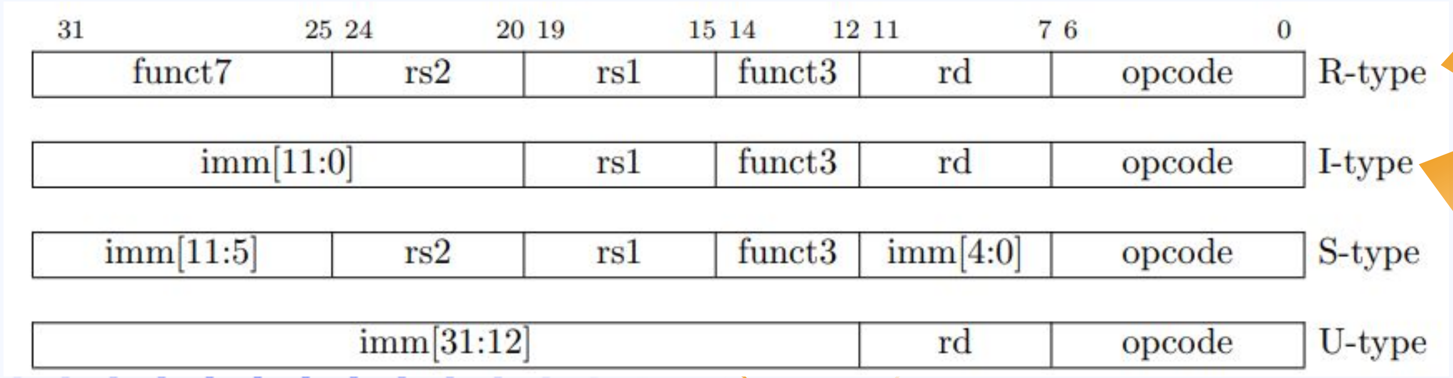
? ? ?

If an arithmetic/logical instruction has three operands, how do you add a constant to a register?

Immediates + instruction formats

Immediates: constants that are encoded as part of an instruction

- **Separate instructions** from register-register (ADD vs ADDI)
- In RISC-V, register-immediate instructions have rd, rs1, and immediate operand



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[31:12]				rd	opcode	U-type

register/register instrs

register/immediate instrs

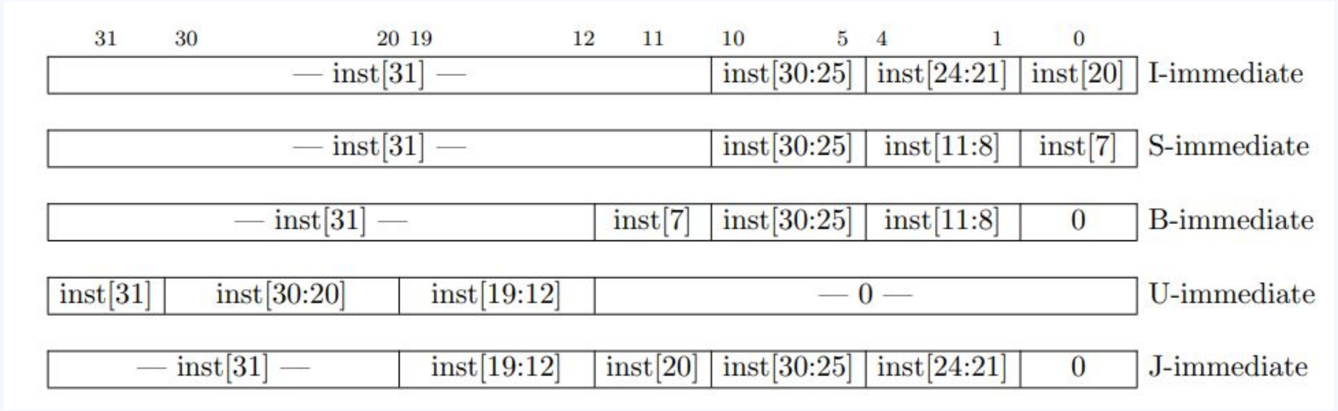
Register-register and Register-immediate instrs

Why no SUBI?

Arithmetic	ADD, SUB	ADDI
Logical	AND, OR, XOR	ANDI, ORI, XORI
Shift	SLL, SRL, SRA	SLLI, SRLI, SRAI
Comparison	SLT[U]	SLTI[U]

```
addi x10, x10, 2 # x10 += 2
xori x12, x11, 0xff # x12 = x11 ^ 0xff
slli x14, x13, 16 # x14 = x13 << 16
```

Immediate encodings



31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

only 12 bits of an I-type instruction are set aside for the immediate

value is *sign-extended* into 32 bits

- need to pay attention to MSB (xori x10, x10, 0xfff will flip all bits, not just the last 12)
- loading a large number into a register may take multiple instructions (hint: LUI)

Role of the assembler

Translation of assembly instruction into machine code is *almost* direct translation

- Unlike compiler, which needs to do things like manage which variables go into which registers, etc.

Assembler also:

- Removes comments
- Translates data and code labels to memory addresses relative to PC
- Synthesizes **pseudo-instructions** (instructions available to compiler but not implemented by microarchitecture)
  - e.g. `mv rd, rs # rd = rs` gets synthesized to `add rd, rs, x0`