



RISC-V Assembly

HW1b comes out today!





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



addition, logical operations

branching (if-expressions/conditionally, unconditional jumps)


managing data (variables, r/w memory, constants)

i/o

Definitions (from P&H Chapter 2)

Instruction: a command computer hardware understands and obeys


Machine language: a binary representation of machine instructions



```
0x00150513
```

Assembler

Assembly language: a symbolic representation of machine instructions



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

Compiler

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

RISC: reduced instruction set computer (contrast with CISC – C stands for complex) makes it more straightforward for us to implement*

Extensions to support different types of computing (multiplication, floating point, data-level parallelism, crypto, embedded...)

Real-world implementations exist (Google Titan M2)

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 examples: ARM, MIPS)

- Contrast with “register-memory” architecture (example: x86)

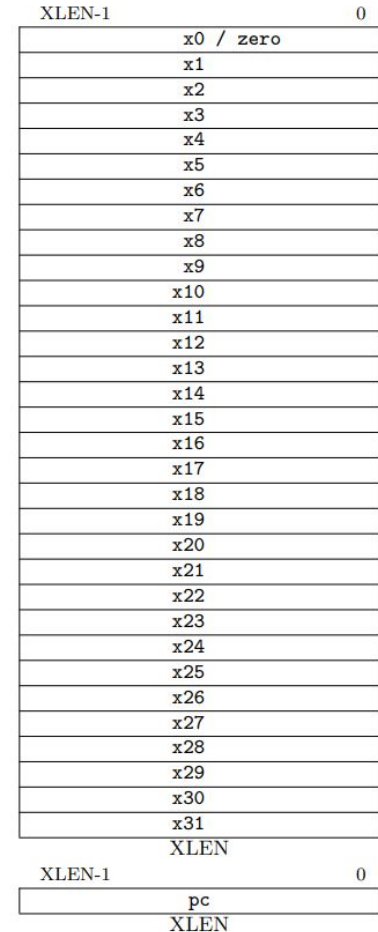
RISC-V Specification

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]

Reminder on signed vs. unsigned: no need for ADDU because 2s complement math just works.

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



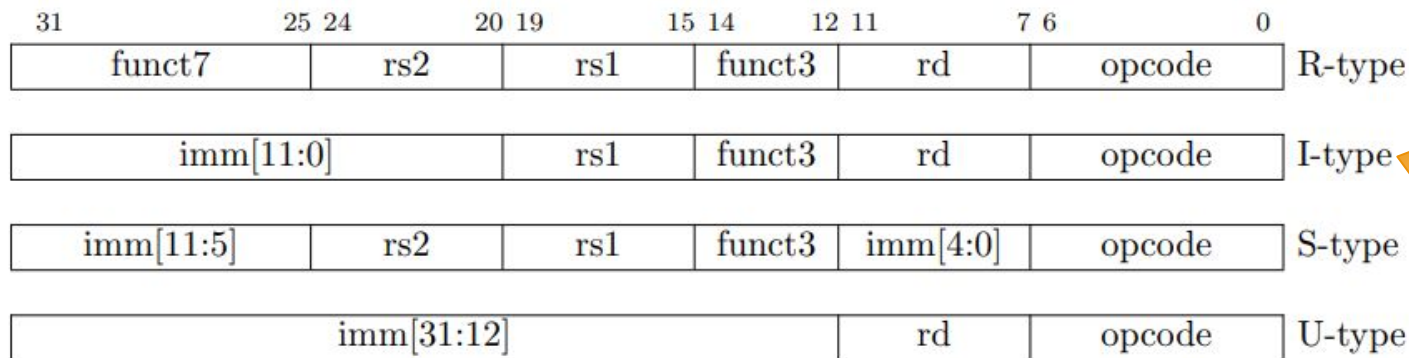
If an arithmetic/logical instruction has three operands, how do you add a constant to a register?

Immediates + instruction formats

Immediates: constants that are encoded as part of an instruction

In RISC-V, register-immediate instructions have rd, rs1, and immediate operand

Instructions come in different *formats*. Specific bits in the instruction (opcodes and funct fields) determine how the CPU should interpret them



register/register instrs

register/immediate instrs

Register-register and Register-immediate instrs

Why no SUBI?

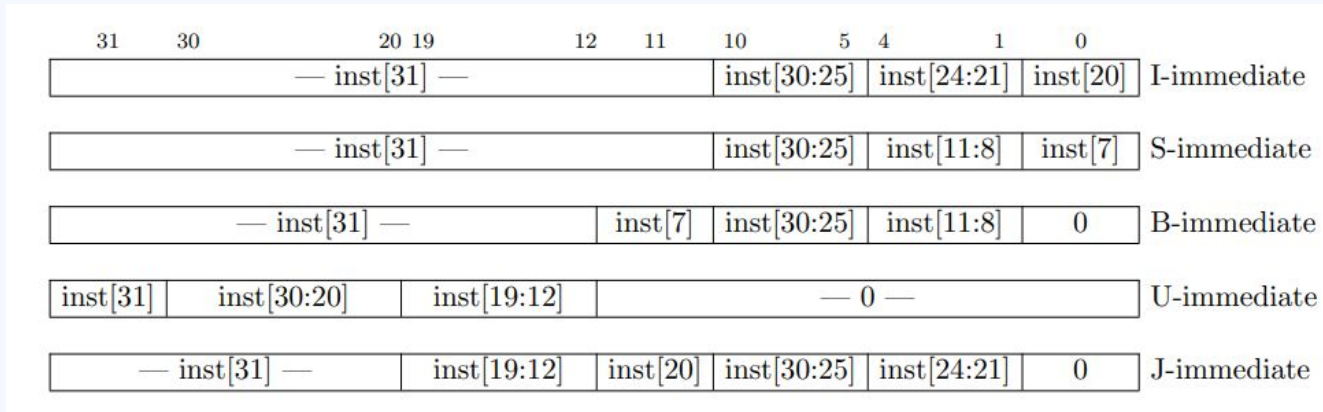
Arithmetic	ADD, SUB	ADDI
Logical	AND, OR, XOR	ANDI, ORI, XORI
Shift	SLL, SRL, SRA	SLLI, SRLI, SRAI
Comparison	SLT[U]	SLTI[U]

```
addi x10, x10, 2 # x10 += 2
```

```
xori x12, x11, 0xff # x12 = x11 ^ 0xff
```

```
sll x14, x13, 16 # x14 = x13 << 16
```

Immediate encodings



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 specified by ISA/available to compiler but not implemented by microarchitecture)

e.g. `mv rd, rs # rd = rs` gets synthesized to `add rd, rs, x0`

Saving registers, the stack

Reasons to save registers: function calls, too many variables for registers

Registers are saved in the *stack* (area of memory reserved for this purpose)

Some ISAs (x86, ARMv7-M) have instructions for pushing and popping and special register to keep track of stack

In RISC-V: compiler or assembly programmer responsible for managing stack

```
complexMul:
    # Place the 4 input arguments and return address on the stack
    addi sp, sp, -28
    sw x0, 24(sp) # tmp. res 2
    sw x0, 20(sp) # tmp. res 1
    sw ra, 16(sp) # return address
    sw a0, 12(sp) # a
    sw a1, 8(sp)  # b
    sw a2, 4(sp)  # c
    sw a3, 0(sp)  # d
```

Ripes complexMul example

Stores are S-type instructions

$s\{d|w|h|b\} rs2, imm(rs1)$

$Mem[rs1 + imm] = rs2$

Loads are I-type

$l\{d|w|h|b\}[u] rd, imm(rs1)$

$rd = Mem[rs1 + imm]$

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 *next week*

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

Stored-program computers

Modern computers hinge on two principles:

- Instructions are represented in memory the same way as numbers
- Memory can be altered by programs

First principle means that:

- Instructions live in memory
- The CPU needs to have a way of interpreting an instruction, just as it would any other data in memory

Reference slide: programs in memory

Object file contains different segments (compiler knows the addresses of these in memory)

data segment: initialized static and global vars

code/text segment: code

bss segment: uninitialized static and global vars

heap and stack: typically come after bss, grow towards each other. Heap is for dynamically allocated memory. Stack is for return addresses, saving registers, automatic variables

See Ripes [complexMul](#) example for use of data and text segments