2025-10-23 - Microcomputers

Volatile

- Any variable can be declared volatile.
  - `volatile int c`
  - `volatile int *pc;`
- Tells C compiler never to trust recently-accessed value for the variable; always re-read memory to get latest value.

Ex.

```
d = a + b * c + b / c;
```

- Most compilers only read 'b' from memory once on any subsequent appearances for the variable 'b'
- When 'c' is declared volatile, C compiler must read the value of 'c' from memory every time it apears in the program; it is not allowed to re-use a recently obtained value.

SimpleCpu ISA

3 types of instructions

Arithmetic:

```
ADD rA, rB
SUB rA, rB
MUL rA, rB
AND rA, rB
```

Data Movement:

```
MOV rA, rB // rA = rB
MOV rA, IMM // rA = IMM
```

$\text{IMM} \in \{-1, 1\}$

Memory access and Output

```
LOAD rA, [rB] // rA = Mem[rB]
STORE rA, [rB] // Mem[rB] = rA, destination of rB assigned to rA
```

$0 - 127 \rightarrow \text{memory}$
$\geq 128 \rightarrow \text{output reg (hex display)}$

Hardware:

$$\text{[1:0] Data} \rightarrow \text{ALU (SW[4:3] to control operations)} \rightarrow \text{LED[5:0]}, \text{outputs}$$

SW[4:3]:

- 00 $\rightarrow$ +
- 01 $\rightarrow$ -
- 10 $\rightarrow$ $*$
- 11 $\rightarrow$ &

LED[5:0]:

-

Register file:

SW[9]: wA (decoder / enabler)
- 0: write to memory, do not write to any Register
- 1: write to register, designated by rA

SW[8:7]: rA
SW[6:5]: rB

Mux: SW[1:0]: Display result
- 01: for arithmetic operations

ex. `MUL r0, r3`

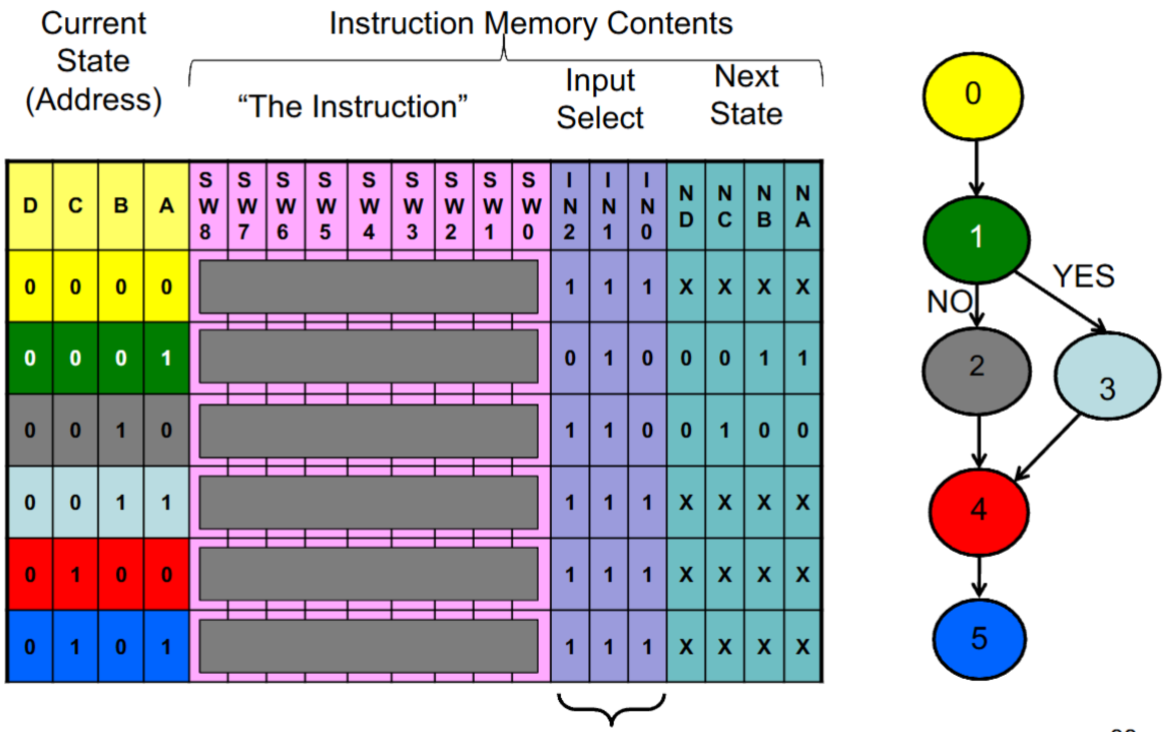
- SW[9]: 1
- SW[8:7]: 00
- SW[6:5]: 11
- SW[4:3]: 10 (ALUop)
- SW[1:0]: 01

Instruction Sequence
- Given a starting address and load signal
- Address sent to Instruction Memroy

Instruction memory stores "instruction words" ie. bits that control the CPU circuit (controls SW[9:0])
- Next instruction is stored at "PC+1"
- User counter to generate PC, PC+1, PC+2

If/else:
- Branches
  - `Bxx LOCATION`
  - Bxx = branch on condition xx, one of $\{C, B, VA, VS, N, Z\}$
- Jumps
  - `JMP LOCATION //always go to location`

2025-10-23 - RISC-V ISA



D	C	B	A	SW8	SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0	IN2	IN1	IN0	ND	NC	NB	NA
0	0	0	0										1	1	1	X	X	X	X
0	0	0	1										0	1	0	0	0	1	1
0	0	1	0										1	1	0	0	1	0	0
0	0	1	1										1	1	1	X	X	X	X
0	1	0	0										1	1	1	X	X	X	X
0	1	0	1										1	1	1	X	X	X	X

010=V+ (branch if V+), 110=1 (jump), 111=0 (do not jump, go to current+1) [38]

- For input select = [1, 1, 1], move to next instruction. (Next state = [x,x,x,x])
- For state 1, if overflow = 1, next state -> 3. Else + 1 (next state -> 2)
- For state 2, we want it only to go to next state -> 4

Chapter 6 - RISC-V ISA

- To negate a, it is essentially $0 - a$, so most instructions only provide subtraction instructions, not negation.

Architecture: Programmer's view of computer (defined by instructions, operand locations).

Microarchitecture: How to implement an architecture in hardware.

Assembly Language: Human-readable (perhaps) format of computer instructions.

Machine Language: Computer-readable format (binary)

RISC-V Assembly:

Addition:

```
# s0 = a, s1 = b, s2 = c
add a, b, c # b and c are operated on, a is the result.
```

Subtraction:

```
sub a, b, c
```

Simplicity favors regularity

- Consistent instruction format.
- Same number of operands (two sources, one destination)
- Easier to encode and handle in hardware

Ex. a = b + c - d

In assembly:

```
add t, b, c # t = b + c
sub a, t, d # a = t - d
```

Operands

Operand location: Physical location on hardware ie Registers, memory, constants.

Registers are faster than memory.

“32-bit architecture”: Uses 32-bit registers, operates on 32-bit data.

Name	Register Number	Usage
`zero`	x0	Constant value 0
`ra`	x1	Return address
`sp`	x2	Stack pointer
`gp`	x3	Global pointer
`tp`	x4	Thread pointer
`t0-2`	x5-7	Temporaries
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`a0-1`	x10-11	Function arguments / return values
`a2-7`	x12-17	Function arguments
`s2-11`	x18-27	Saved registers
`t3-6`	x28-31	Temporaries

- x0 is always 0
- Jump instruction can only write to x1

Registers

- Can use name (`ra, zero` or `x0, x1`, etc.)
- Convention:
  - `s0 - s11`: holds variables
  - `t0 - t6`: hold temporary variables

Can use immediate instructions.

ex. addi

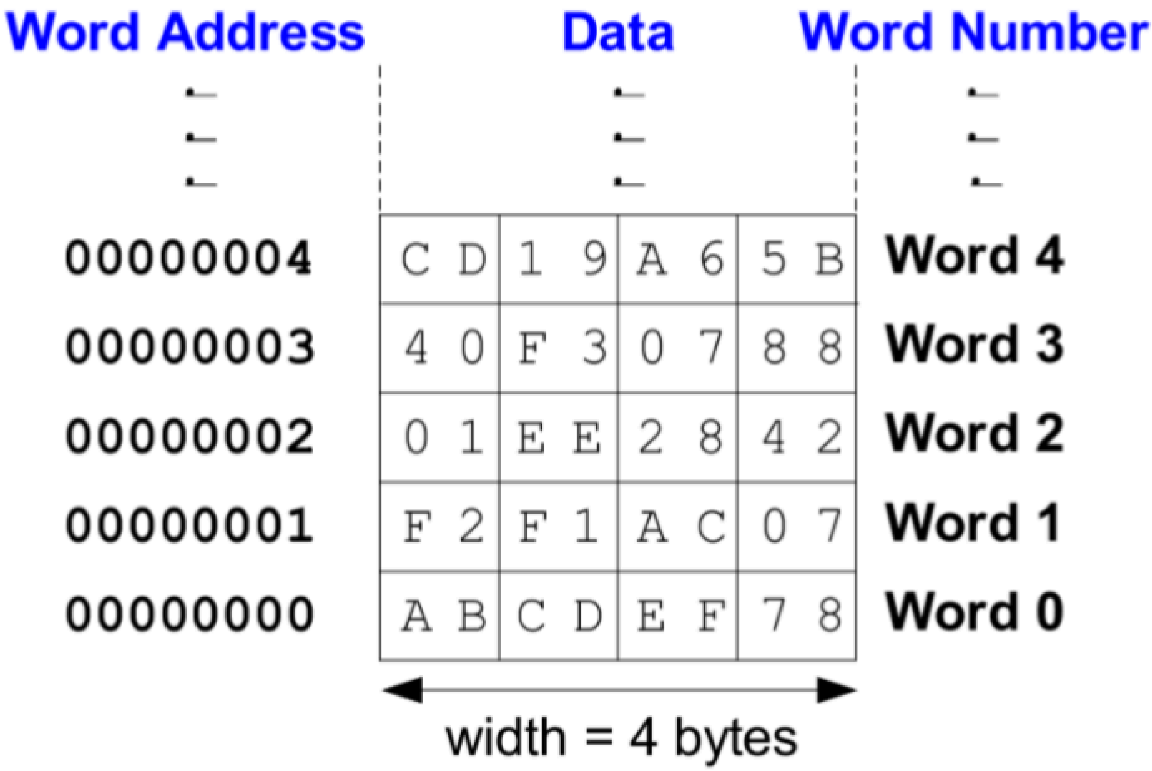
```
# s0 = a, s1 = b
addi s0, s1, 6
```

Memory

- Much slower than registers.
- Too much data to fit into 32 registers
- Commonly used variables kept in registers
- Cache: faster memory with smaller size in between memory and register speed

Word-addressable memory:

- Each 32-bit data word has a unique address



- To read from memory, call “load word” = `lw`

```
lw t1, 5(s0)
lw destination, offset(base)
```

- Add base address (s0) to the offset (5)
- address = (s0 + 5)
- t1 holds the data value at address (s0 + 5)
- *any register* may be used as base address

ex. Read word of data at memory address 1 into s3.

```
lw s3, 1(zero) # read memory word 1 into s3
```

- To write to memory, call “store” = `sw`

ex. Write value in t4 to memory address 3

```
sw t4, 0x3(zero) # write value in t4 to memory word 3
```

- Add base address (zero) to offset (0x3)
- Address: (0 + 0x3) = 3
- Result: Writes the

Byte-addressable Memory

- Each data byte has unique address

- Load/store words or single bytes: load byte `lb` and store byte `sb`
- 32-bit word = 4 bytes, so word address *increments by 4*

Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	C D 1 9 A 6 5 B	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0
MSB			LSB		width = 4 bytes	

- **Example:** store the value held in `t7` into memory address 0x10 (16)
  – if `t7` holds the value 0xAABBCCDD, then after the `sw` completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero)   # write t7 into address 16
```

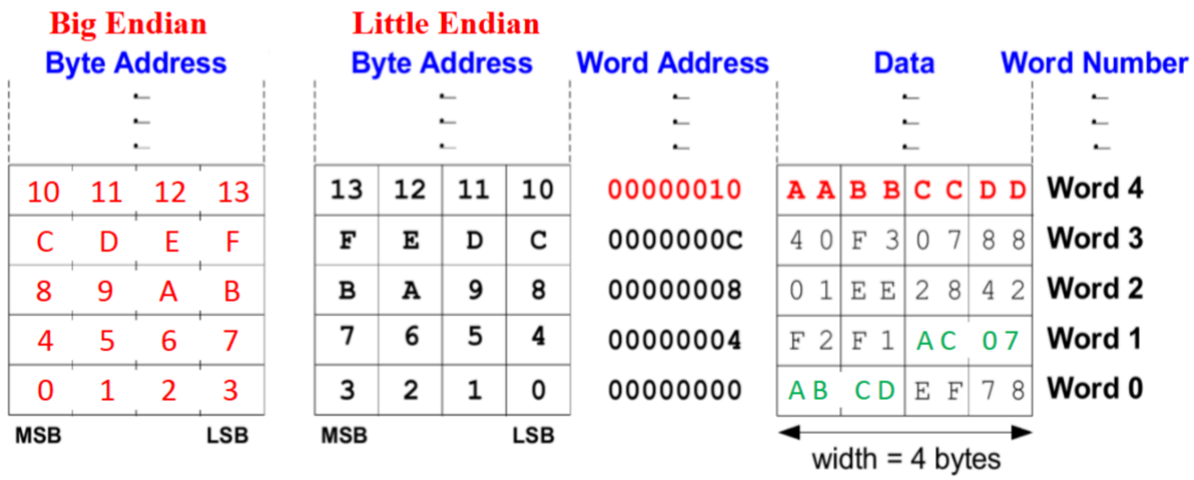
Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	A A B B C C D D	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0
MSB			LSB		width = 4 bytes	

- Alignment: word address is not multiple of 4

  `lw t1, 2(zero) // 0xAC_07_AB_CD`

- That is Little-Endian, what about Big-Endian?

  `lw t1, 2(zero) // 0xEF_78_F2_F1`



Constants

12-bit signed constants (immediates) using addi:

Assembly:

```
# so = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

C:

```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

However, for assembly this does not work for values over 12 bits.

For these cases, use `lui` and addi.

- `lui` puts an immediate in the upper 20 bits of destination and 0s in lower 12 bits

Assembly:

```
# s0 = a
lui s0, 0xFEDC8
addi s0, s0, 0x765
```

C:

```
int a = 0xFEDC8765;
```

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in `lui`

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9      # s0 = 0xFEDC9000
addi s0, s0, -341     # s0 = 0xFEDC9000 + 0xFFFFFEAB
                      #    = 0xFEDC8EAB
```

Logic/Shift instructions

- and: useful for bit masking. ex.
  - 0xF234012F AND 0X000000FF = 0x0000002F
  - masks the last 2 bytes
- or: useful for combining bit fields
  - Combine 0xF2340000 with 0x000012BC:
  - 0xF2340000 OR 0X000012BC = 0XF23412BC
- xor: uesful for invergin bits:
  - A XOR −1 = NOT A (-1 = 0xFFFFFFFF)
- sll: shift left logical
  - slli: immediate (`slli t0, t1, 23` $\Longrightarrow$ t0 = t1 << 23)
- srl: shift right logical (Inserts zeroes)
  - srli: immediate
- sra: shift right arithmetic (Preserves sign bit)
  - srai: immediate

for non-immediate shifts, ie `sll t0, t1, t2` only takes the least 5 significant bits of t2.

Multiplication and Division

32 x 32 multiplication $\rightarrow$ 64 bit result

```
mul s3, s1, s2 # s3 = lower 32 bits of result
mulh s4, s1, s2 # s4 = higher 32 bits of result
```

32-bit division $\rightarrow$ 32 bit quotient and remainder

- div s3, s1, s2
- rem s4, s1, s2

Branching

Conditional branches:

- beq: branch if equal
- bne: branch if not equal

- blt: branch if less than
- bge: branch if greater than or equal

Unconditional:
- j: jump
- jr: jump register
- jal: jump and link
- jalr: jump and link register

```
# RISC-V assembly
 addi s0, zero, 4        # s0 = 4
 addi s1, zero, 1        # s1 = 1
 slli s1, s1, 2          # s1 = 1 << 2 = 4
 beq  s0, s1, target     # branch is taken
 addi s1, s1, 1          # not executed
 sub  s1, s1, s0         # not executed

target:                  # label
 add  s1, s1, s0         # s1 = 4 + 4 = 8
```

Labels indicate instruction location (or data location). They can't be reserved words and must be followed by a colon (:)

```
# RISC-V assembly
  j       target          # jump to target
  srai    s1, s1, 2       # not executed
  addi    s1, s1, 1       # not executed
  sub     s1, s1, s0      # not executed

 target:
  add     s1, s1, s0      # s1 = 1 + 4 = 5
```

Conditional Statements and Loops

If statement:

C:

```c
if (i == j) {
    f = g + h;
}
f = f - i;
```

RISC-V assembly:

```
# s0 = f, s1 = g, s2 = h, s3 = i, s4 = j
bne s3, s4, L1
add s0, s1, s2

L1:
    sub s0, s0, s3
```

If else

C:

```c
if (i == j) {
    f = g + h;
} else {
    f = f - i;
}
```

Assembly:

```
# s0 = f, s1 = g, s2 = h, s3 = i, s4 = j
bne s3, s4, L1
add s0, s1, s2
j done

L1:
    sub s0, s0, s3
done:
```

While loop

C:

```c
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

Assembly:

```
#s0 = pow, s1 = x

addi s0, zero, 1
add s1, zero, zero
addi t0, zero, 128

while:
    beq s0, t0, done
```

```
    slli s0, s0, 1
    addi s1, s1, 1
    j while
done:
```

For loop for (initialization; condition; loop operation) { statement }

C:

```
int sum = 0;
int i;

for (i = 0; i != 10; i++) {
    sum += i;
}
```

Assembly:

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    add s0, zero, zero
    addi t0, zero, 10
for:
    beq s0, t0, done
    add s1, s1, s0
    addi s0, s0, 1
    j for
done:
```

Less Than

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
        addi   s1, zero, 0
        addi   s0, zero, 1
        addi   t0, zero, 101
loop:
        slt    t2, s0, t0
        beq    t2, zero, done
        add    s1, s1, s0
        slli   s0, s0, 1
        j      loop
done:
```

slt: set if less than instruction
`slt t2, s0, t0` # if s0 < t0, t2 = 1
otherwise t2 = 0

2025-10-30

More on Jumps and Pseudoinstructions

Jumps

- Two types of unconditional jumps
  - Jump and link (jal, rd, imm)
    - rd = PC+4; PC = PC + imm
  - Jump and link register (jalr rd, rs, imm)
    - rd = PC+4; PC = [rs] + SignExt(imm)
    - jump to specified register, can add a small constant after

Pseudoinstructions

- Assembler converts to real instructions

```
j im    # jal x0, imm. Jump to some place without returning
jal imm # jal ra, imm. Allows you to omit ra
jr rs   # jalr x0, rs, 0.
ret     # jalr x0, ra, 0. (jr ra)
```

Labels

- Indicate where to jump
- Represnted in jump as immediate offset

imm = # bytes past immediate offset

- eg. imm = (51C - 300) = 0x21C
- jal ra 0x21C

Long Jumps

- Immediate is limited in size
  - 20 bits for jal, 12 bits for jalr

- Limits how far program can jump
- Special instruction to help jumping further
  - `auipc rd, imm`

Pseudoinstruction	RISC-V Instructions
j label	jal zero, label
jr ra	jalr zero, ra, 0
mv t5, s3	addi t5, s3, 0
not s7, t2	xori s7, t2, -1
nop	addi zero, zero, 0
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF
bgt s1, t3, L3	blt t3, s1, L3
bgez t2, L7	bge t2, zero, L7
call L1	auipc ra, $imm_{31:12}$ jalr ra, ra, $imm_{11:0}$
ret	jalr zero, ra, 0

Useful pseudoinstructions:
- mv (move from one register to another)
- call

Machine Language
- Every instruction is 32 bits
- Binary

4 types of instruction formats:

R-Type
- Source registers: rs1, rs2
- Destination register: rd
- Operation: op
- Function: funct7, funct3

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-Type

operands:

- register source: rsl
- register destination: rd
- 12-bit two's complement immediate: imm

other:

- opcode: op
- 3-bit function code: funct3

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

S/B Type

- Store type
- Branch type – does not use imm_0 (always 0)
- Only differ in 'immediate' encoding

S Type

Store type 3 operands:

- rsl: base register
- rs2: value to be stored
- imm: 12-bit two's complement immediate

other:

- op: opcode
- funct3: 3 bit function code

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

B type

Branch type

3 operands:

- rsl: base register
- rs2: value to be stored
- $\mathrm{imm}_{12:1}$: 12-bit two's complement immediate

other:

- op: opcode
- funct3: 3 bit function code

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

U type

- ***Upper-immediate-Type***
- Used for load upper immediate (`lui`)
- 2 operands:
  - `rd`: destination register
  - $imm_{31:12}$:upper 20 bits of a 32-bit immediate
- Other fields:
  - `op`: the *operation code* or *opcode* – tells computer what operation to perform

U-Type

31:12	11:7	6:0
$imm_{31:12}$	rd	op
20 bits	5 bits	7 bits

J Type

- ***Jump-Type***
- Used for jump-and-link instruction (`jal`)
- 2 operands:
  - `rd`: destination register
  - `imm`$_{20,10:1,11,19:12}$: 20 bits (20:1) of a 21-bit immediate
- Other fields:
  - `op`: the operation code or opcode – tells computer what operation to perform

J-Type

31:12	11:7	6:0
imm$_{20,10:1,11,19:12}$	rd	op
20 bits	5 bits	7 bits

- Note: `jalr` is I-type, not j-type, to specify rs1

read pg 70-75

Assembly

- R-type (3 registers)

  `add rd, rs1, rs2`

  also: sub, mul, div, rem, and, or, xor, sll, srl, sra, slt, sltu
- I-type (2 registers + constant)

  `addi rd, rs1, Imm12`

  also: addi, andi, ori, xori, jalr, lb/lbu/lh/lhu/lw, sb/sh/sw, slti, sltiu, beq, bne, blt, bltu, bge, bgeu
- J-type (1 register + constant)

  `jal rd, Imm20`

  also: lui, auipc

Registers

There are 32 registers. x0-x31
- x0 is always 0.
- t0-t6 are temporary registers
- s0-s11 save-to-use registers
- a0-a7 function arguments, a0-a1 results
- special purpose:
  - x0: zero
  - x1: ra
  - x2: sp
  - x3: gp (global pointer)
  - x4: tp (for multithread)
  - x8: fp (frame pointer)

Accessing Memory

Read
- `lw t3, 0x04(t5)`
- copies memory value to register t3
- address in memory is 0x04 + t5

Write
- `sw t3, 0x04(t5)`
- copies register t3 value to memory
- address in memory is 0x04 + t5

Assembler Process takes assembly file (.s) → (as program.s -g -o program.o) .o file → ld program.o --no-relax -o program.exe → program.exe

- Two passes
  - 1: find or compute values for all symbols (constants)
  - 2: replace all symbols with binary value and output machine Language

- We use symbols (names) to represent constants (numbers) or constant expressions
- Can also type commands to assembler through directives, . commands eg: ".global" which assigns a global variable (can be accessed from other files)
- Assembler gives pseudoinstructions which map to 1 or 2 real instructions

```
.equ # symbolic constant, name -> value
.text # beginning of instructions
ecall # call operating system

# Ending a program

li a0, 0 # output 0 when proceeding to close program, no errors
li a7, LX_EXIT # exit the program
ecall
```

Example program

Problem:

- array A[*] = {5, 3, −6, 19, 8, 12}
- A is located in memory at 0x100
- add up all elements of array into t1

```
.equ LX_EXIT, 93

.text
_start:
add t1, zero, zero # sum
addi t5, zero, 0x100 # A
addi t3, zero, 6 # N

loop:
    beq t3, zero, done
    lw t2, 0(t5) # 5
    add t1, t1, t2
    addi t5, t5, 4 # addr += 4
    addi t3, t3, -1
    j loop
done:
    mv a0, 0
    mv a7, LX_EXIT
    ecall

.data

A:
    .word 5, 3, -6, 19, 8, 12
    Aend:
N:
    .word (Aend-A)/4
```

2025-11-04 Bare Metal Programming

Operating system:

- Control access to shared resources
- memory, CPU, disk, screens, etc.
- allow multiple programs to share resources

Bare metal programming:

- No OS
- Not good for sharing resources between programs

LEDR and SW

LEDR:

- Address 0xFF200000: only last 10 bits are used

SW:

- Address: 0xFF2000040: only last 10 bits are used

```
# read switches [9:0]
la t0, SW_BASE # addr. 0XFF2000040
lw t1, 0(t0)
# write LEDR[9:0]
la t2, LEDR_BASE # addr. 0xFF200000
sw t1, 0(t2)
# Continuously copy SW to LEDR
loop:
  lw t1, 0(t0)
  sw t1, 0(t2)
  j loop
```

KEY[1:0]

- Can directly read KEYS
- Can configure to send "interrupts" to CPU

Time Delay

- Do nothing: `nop`

```
# Program to blink LED with time delay here
```

HEX Display

VGA Output

JTAG UART

- can print a character on the screen

```
.equ JTAG_UART_BASE, 0xFF201000

.global _start
_start:
  la t0, JTAG_UART_BASE
  li t1, 'C'
  sw t1, 0(t0)
  li t1, 'P'
  sw t1, 0(t0)
  li t1, 'E'
  sw t1, 0(t0)
```

```
  li t1, 'N'
  sw t1, 0(t0)
stop:
  j stop
```