E85

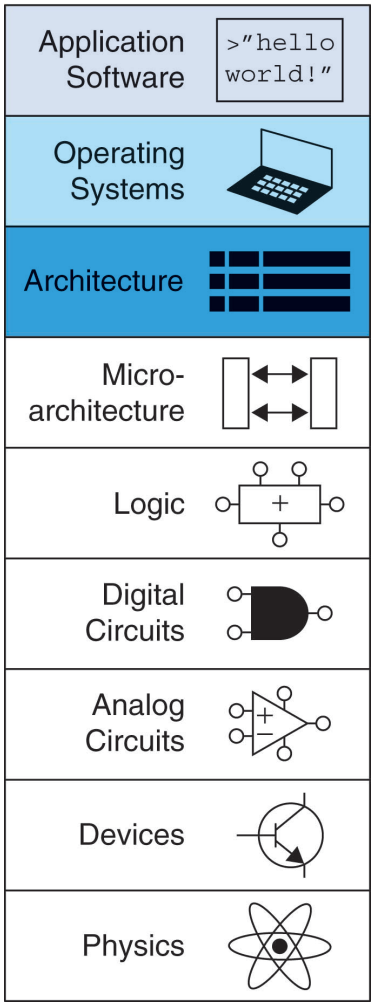
Digital Electronics & Computer Architecture

Lecture 18:

RISC-V Machine Language

Lecture 18

- Machine Language
  - Instruction types and formats
  - Interpreting machine code
  - Addressing modes
  - The stored program
  - Odds & Ends
    - Pseudoinstructions
    - Signed and unsigned operations
    - Floating point instructions

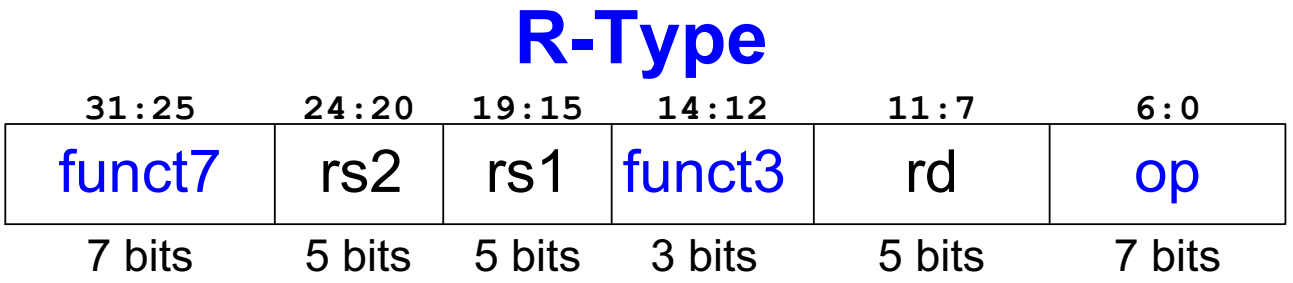


Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- **4 Types of Instruction Formats**
  - R-Type
  - I-Type
  - S/B-Type
  - U/J-Type

R-Type

- *Register-type*
- 3 register operands:
  - `rs1`, `rs2`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode*
  - `funct7`, `funct3`:
    the *function* (7 bits and 3-bits, respectively)
    with opcode, tells computer what operation to perform



R-Type Examples

Assembly	funct7	rs2	rs1	funct3	rd	op	Assembly
	Field Values						
`add s2, s3, s4`	0	20	19	0	18	51	`add x18, x19, x20`
`sub t0, t1, t2`	32	7	6	0	5	51	`sub x5, x6, x7`
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Machine Code

funct7	rs2	rs1	funct3	rd	op	
0000 000	10100	10011	000	10010	011 0011	`(0x01498933)`
0100 000	00111	00110	000	00101	011 0011	`(0x407302B3)`
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Note the order of registers in the assembly code:

```
add rd, rs1, rs2
```

More R-Type Examples

Assembly	funct7	rs2	rs1	funct3	rd	op	Assembly
`sll s7, t0, s1`	0	9	5	1	23	51	`sll x23, x5, x9`
`xor s8, s9, s10`	0	26	25	4	24	51	`xor x24, x25, x26`
`srai t1, t2, 29`	32	29	7	5	6	19	`srai x6, x7, 29`
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Field Values

Machine Code

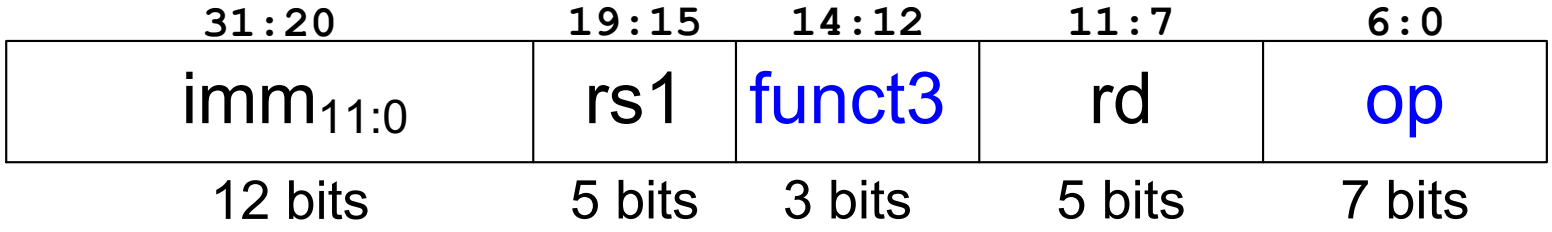
funct7	rs2	rs1	funct3	rd	op	
0000 000	01001	00101	001	10111	011 0011	`(0x00929BB3)`
0000 000	11010	11001	100	11000	011 0011	`(0x01ACCC33)`
0100 000	11101	00111	101	00110	001 0011	`(0x41D3D313)`
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Note immediate shift instructions (like `srai t1, t2, 29`) are R-type instructions. They encode the shift amount in the **rs2** field.

I-Type

- *Immediate-type*
- 3 operands:
  - `rs1`: register source operand
  - `rd`: register destination operand
  - `imm`:12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the *function* (3-bit function code)
    - with opcode, tells computer what operation to perform

I-Type



I-Type Examples

Assembly	$imm_{11:0}$	rs1	funct3	rd	op	Assembly
`addi s0, s1, 12`	12	9	0	8	19	`addi x8, x9, 12`
`addi s2, t1, -14`	-14	6	0	18	19	`addi x18, x6, -14`
`lw t2, -6(s3)`	-6	19	2	7	3	`lw x7, -6(x19)`
`lh s1, 27(zero)`	27	0	1	9	3	`lh x9, 27(x0)`
`lb s4, 0x1F(s4)`	0x1F	20	0	20	3	`lb x20, 0x1F(x20)`
	12 bits	5 bits	3 bits	5 bits	7 bits	

Field Values

Machine Code

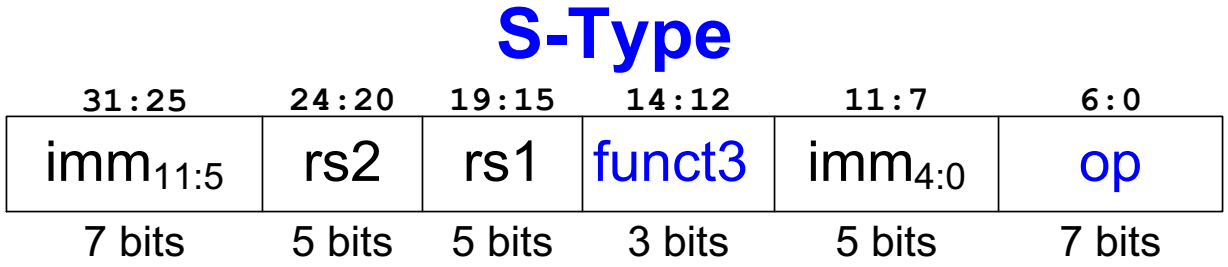
Note the differing order of operands in assembly and machine codes:

```
addi rd, rs1, imm
lw   rd, imm(rs1)
```

$imm_{11:0}$	rs1	funct3	rd	op	
0000 0000 1100	01001	000	01000	001 0011	`(0x00C48413)`
1111 1111 0010	00110	000	10010	001 0011	`(0xFF230913)`
1111 1111 1010	10011	010	00111	000 0011	`(0xFFA9A383)`
0000 0001 1011	00000	001	01001	000 0011	`(0x01B01483)`
0000 0001 1111	10100	000	10100	000 0011	`(0x01FA0A03)`
12 bits	5 bits	3 bits	5 bits	7 bits	

S-Type

- *Store-Type*
- 3 operands:
  - `rs1`: base register
  - `rs2`:value to be stored to memory
  - `imm`:12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the *function* (3-bit function code)
    - with opcode, tells computer what operation to perform



S-Type Examples

Assembly

Field Values

Assembly	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
`sw t2, -6(s3)`	1111 111	7	19	2	11010	35	`sw x7, -6(x19)`
`sh s4, 23(t0)`	0000 000	20	5	1	10111	35	`sh x20, 23(x5)`
`sb t5, 0x2D(zero)`	0000 001	30	0	0	01101	35	`sb x30, 0x2D(x0)`
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Machine Code

Note the differing order of operands in assembly and machine codes:

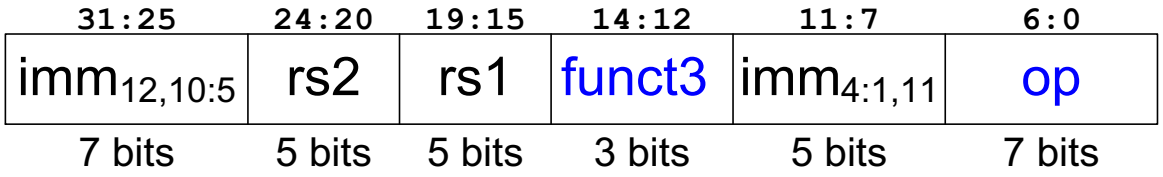
`sw rs2, imm(rs1)`

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
1111 111	00111	10111	010	11010	010 0011	`(0xFE7BAD23)`
0000 000	10100	00101	001	10111	010 0011	`(0x01429BA3)`
0000 001	11110	00000	000	01101	010 0011	`(0x03E006A3)`
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

B-Type

- *Branch-Type* (similar format to S-Type)
- 3 operands:
  - `rs1`: register source 1
  - `rs2`:register source 2
  - $\texttt{imm}_{12:1}$: 12-bit two's complement immediate – address offset
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the *function* (3-bit function code)
    - with opcode, tells computer what operation to perform

B-Type



B-Type Example

- The 13-bit immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- Example:

```
# RISC-V Assembly
  beq s0, t5, L1
  add s1, s2, s3
  sub s5, s6, s7
  lw  t0, 0(s1)
L1:
  addi s1, s1, -15
```

1
2
3
4

L1 is 4 instructions (i.e., **16 bytes**) past beq

imm = 16	0		0	0	0	0		0	0	0	1		0	0	0	0
bit number	12		11	10	9	8		7	6	5	4		3	2	1	0

B-Type Example

Assembly

`beq s0, t5, L1`

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

`beq x8, x30, L1`

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

`(0x01E40863)`

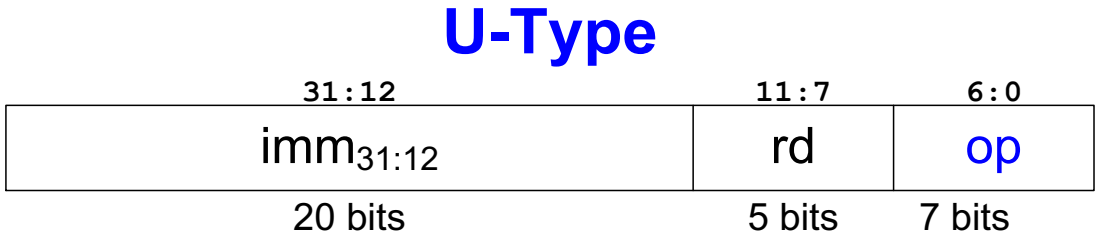
Note the differing order of operands in assembly and machine codes:

`beq rs1, rs2,` $imm_{12:1}$

imm = 16	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

U-Type

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

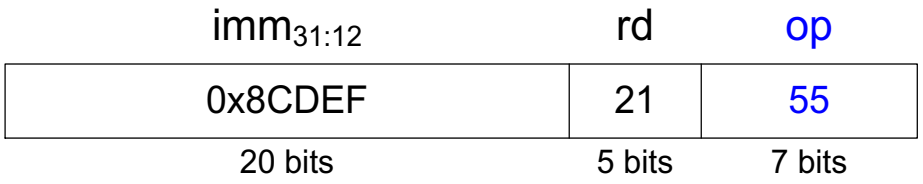


U-Type Example

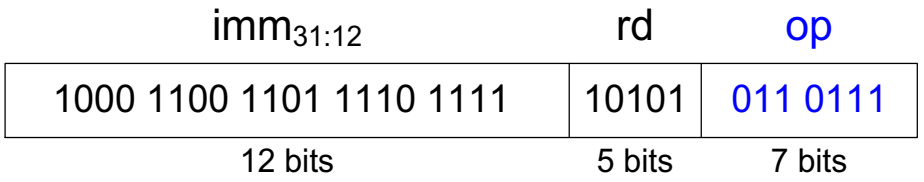
Assembly

```
lui s5,  0x8CDEF
(lui x21, 0x8CDEF)
```

Field Values



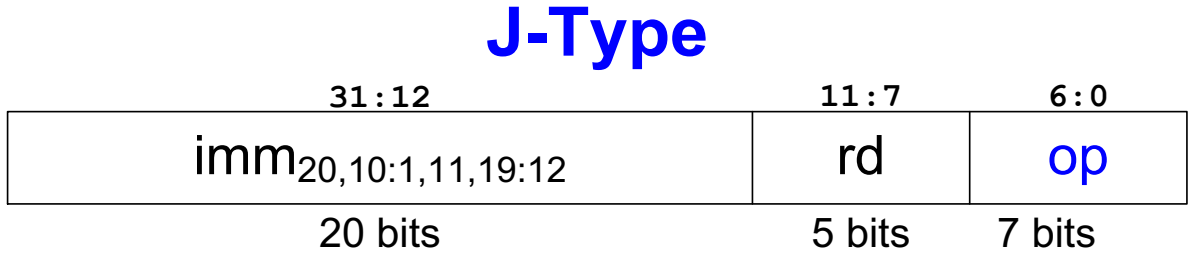
Machine Code



(0x8CDEFAB7)

J-Type

- *Jump-Type*
- Used for jump-and-link instruction (`jal`)
- 2 operands:
  - `rd`: destination register
  - $\texttt{imm}_{20,10:1,11,19:12}$: 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 Example

- Immediate encoding is strange
- **Example:**

```
# Address    RISC-V Assembly
0x0000540C          jal ra, func1
0x00005410          add s1, s2, s3
...                 ...

0x001ABC04  func1:  add s4, s5, s8
...                 ...
```

func1 is 0x1A67F8 bytes past **jal**

imm = 0x1A67F8	1		1	0	1	0		0	1	1	0		0	1	1	1		1	1	1	1		1	0	0	0
bit number	20		19	18	17	16		15	14	13	12		11	10	9	8		7	6	5	4		3	2	1	0

J-Type Example

- Immediate encoding is strange
- **Example:**

Assembly

```
jal ra, func1
(jal x1, func1)
```

Field Values

$imm_{20,10:1,11,19:12}$	rd	op
1111 1111 1000 1010 0110	1	111
20 bits	5 bits	7 bits

Machine Code

$imm_{20,10:1,11,19:12}$	rd	op	
1111 1111 1000 1010 0110	00001	110 1111	(0xFF8A60EF)
12 bits	5 bits	7 bits	

imm = 0x1A67F8	1		1	0	1	0		0	1	1	0		0	1	1	1		1	1	1	1		1	0	0	0
bit number	20		19	18	17	16		15	14	13	12		11	10	9	8		7	6	5	4		3	2	1	0

Unraveling the web of lies: `jr`

- `jr ra` is not a real RISC-V instruction.
- It is a pseudoinstruction for `jalr x0, ra, 0`
- `jalr` is not a J-type instruction.

jalr

- `jalr` is an I-type instruction.
- It writes PC+4 to **rd** and jumps to **rs1+imm**.
- **Example:**

```
lui s7, 0x801FA      # s7 = 0x801FA000
jalr s2, s7, 0x7BC   # s2 = PC + 4
                     # PC = s7 + 0x7BC
                     #    = 0x801FA7BC
```

In this case, **rd** = `s2`, **rs1** = `s7`, **imm** = `0x7BC`

Review: Instruction Formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`, `addi`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediate*ly available from instruction
- 12-bit two's complement number
- `addi`: add immediate
- Subtract immediate (`subi`) necessary?

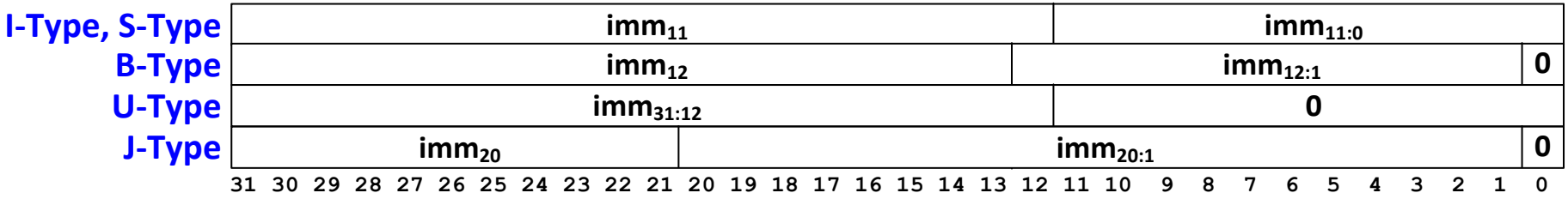
C Code

```
a = a + 4;
b = a - 12;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, s0, 4
addi s1, s0, -12
```

Constants/Immediates



Immediate Encodings

Instruction Bits

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7
R-Type	funct7							4	3	2	1	0	rs1					funct3			rd1				
I-Type	11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd1				
S-Type	11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0
B-Type	12	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	11
U-Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd1				
J-Type	20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd1				

- Immediate bits *mostly* occupy **consistent instruction bits**.
- **Sign bit** of signed immediate is in **msb** of instruction.
- Recall that **`rs2`** of R-type can encode immediate shift amount.

Instruction Fields & Formats

Instruction	op	funct3	Funct7	Type
`add`	0110011 (51)	000 (0)	0000000 (0)	R-Type
`sub`	0110011 (51)	000 (0)	0100000 (32)	R-Type
`and`	0110011 (51)	111 (7)	0000000 (0)	R-Type
`or`	0110011 (51)	110 (6)	0000000 (0)	R-Type
`addi`	0010011 (19)	000 (0)	-	I-Type
`beq`	1100011 (99)	000 (0)	-	B-Type
`bne`	1100011 (99)	001 (1)	-	B-Type
`lw`	0000011 (3)	010 (2)	-	I-Type
`sw`	0100011 (35)	010 (2)	-	S-Type
`jal`	1101111 (111)	-	-	J-Type
`jalr`	1100111 (103)	000 (0)	-	I-Type
`lui`	0110111 (55)	-	-	U-Type

See Appendix B, Table B.2 for other encodings

Interpreting Machine Code

- Write in binary
- Start with **op** (& **funct3**): tells how to parse rest
- **Extract fields**
- **op**, **funct3**, and **funct7** fields to tell operation
- **Ex:** 0x41FE83B3 and 0xFDA58393

0x41FE83B3: 0100 0001 1111 1110 1000 0011 1011 0011

op = 51: R-type

0xFDA48393: 1111 1101 1010 0100 1000 0011 1001 0011

op = 19, funct3 = 0: addi (I-type)

Interpreting Machine Code

- Write in binary
- Start with op (& funct3): tells how to parse rest
- **Extract fields**
- op, funct3, and funct7 fields to tell operation
- **Ex:** 0x41FE83B3 and 0xFDA58393

Machine Code

	funct7	rs2	rs1	funct3	rd	op
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Field Values

funct7	rs2	rs1	funct3	rd	op
32	31	29	0	7	51
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Assembly

```
sub x7, x29, x31
(sub t2, t4, t6)
```

Interpreting Machine Code

- Write in binary
- Start with op (& funct3): tells how to parse rest
- **Extract fields**
- op, funct3, and funct7 fields to tell operation
- **Ex:** 0x41FE83B3 and 0xFDA58393

Machine Code

	funct7	rs2	rs1	funct3	rd	op
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Field Values

funct7	rs2	rs1	funct3	rd	op
32	31	29	0	7	51
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Assembly

```
sub x7, x29, x31
(sub t2, t4, t6)
```

Machine Code

	$imm_{11:0}$	rs1	funct3	rd	op
(0xFDA48393)	1111 1101 1010	01001	000	00111	001 0011
	12 bits	5 bits	3 bits	5 bits	7 bits

Field Values

$imm_{11:0}$	rs1	funct3	rd	op
-38	9	0	7	19
12 bits	5 bits	3 bits	5 bits	7 bits

Assembly

```
addi x7, x9, -38
(addi t2, s1, -38)
```

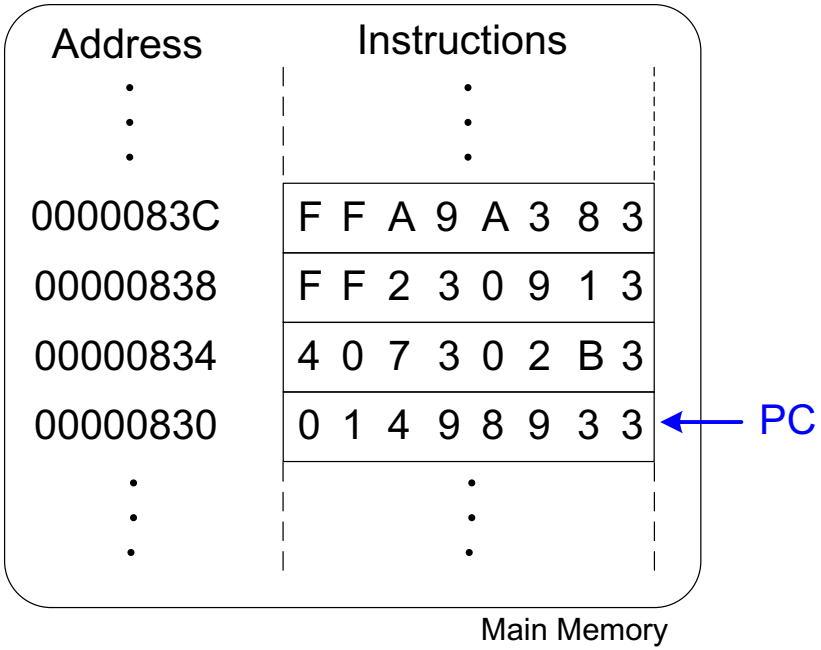
Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
`add  s2, s3, s4`	`0x01498933`
`sub  t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw   t2, -6(s3)`	`0xFFA9A383`

Stored Program



Program Counter (PC): keeps track of current instruction

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

Addressing Modes

Register Only

- Operands found in registers
  - **Example:** `add s0, t2, t3`
  - **Example:** `sub t6, s1, s0`

Immediate

- 12-bit signed immediate used as an operand
  - **Example:** `addi s4, t5, -73`
  - **Example:** `ori  t3, t7, 0xFF`

Addressing Modes

Base Addressing

- Loads and Stores
- Address of operand is:

  `base address + immediate`

  - **Example:** `lw  s4, 72(zero)`
    - address = `0 + 72`

  - **Example:** `sw  t2, -25(t1)`
    - address = `t1 - 25`

Addressing Modes

PC-Relative Addressing: branches and `jal`

Address	Instruction
0x354	L1: addi s1, s1, 1
0x358	sub t0, t1, s7
...	...
0xEB0	bne s8, s9, L1

The label is (0xEB0-0x354) = 0xB5C (**2908**) instructions **before** `bne`

$imm_{12:0}$ = -2908	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
bit number	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Assembly

`beq s8, s9, L1`
`(beq x25, x26, L1)`

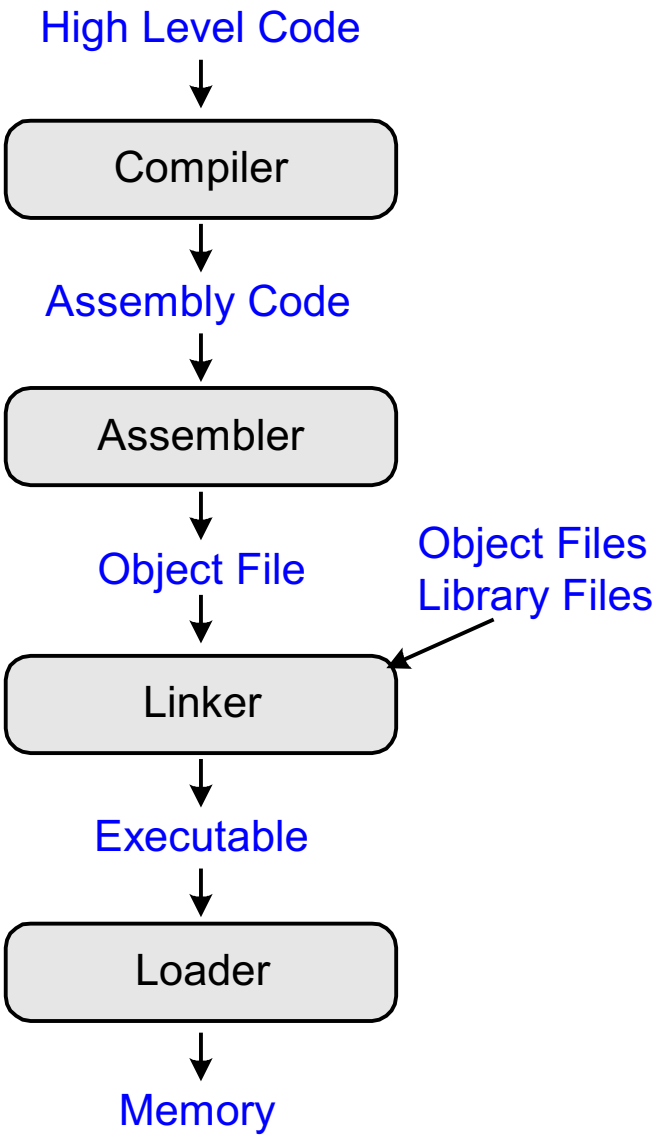
Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	24	25	1	**0010 0**	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
1100 101	11000	11001	001	**0010 0**	110 0011	`(0xCB8C9263)`
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

How to Compile & Run a Program



Grace Hopper, 1906-1992

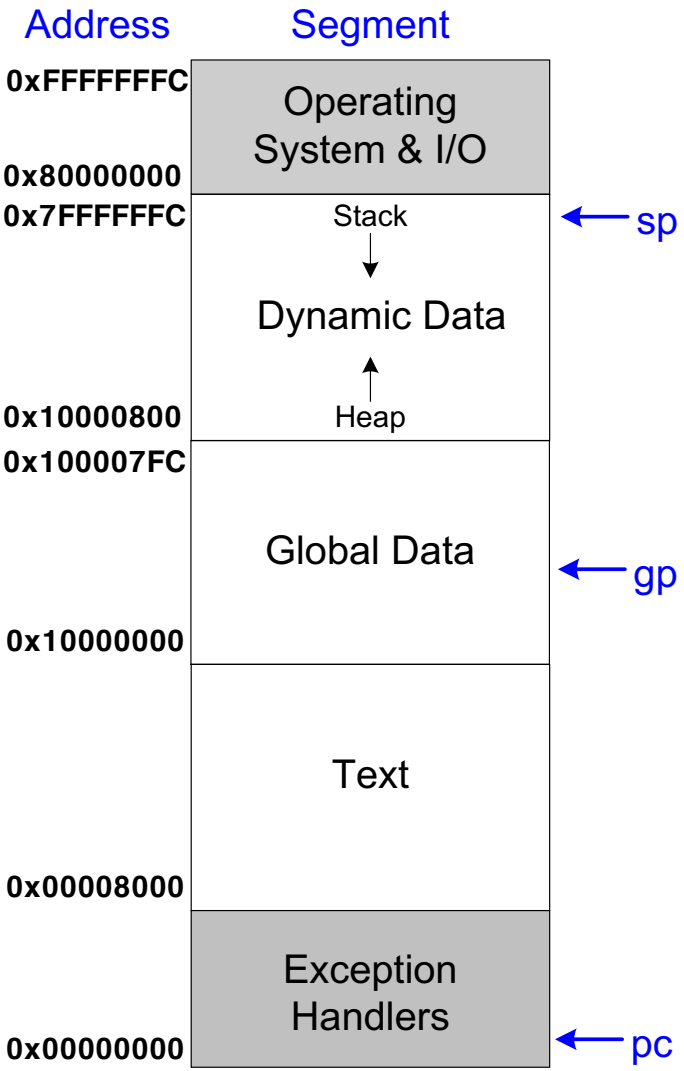
- Graduated from Yale University with a Ph.D. in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others

What is Stored in Memory?

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

- How big is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

Example RISC-V Memory Map



Example Program: RISC-V Assembly

```
int f, g, y;  // global


int main(void)
{


  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi sp, sp, -4    # stack frame
  sw   ra, 0(sp)     # store $ra
  addi a0, zero, 2   # a0 = 2
  sw   a0, f         # f = 2
  addi a1, zero, 3   # a1 = 3
  sw   a1, g         # g = 3
  jal  sum           # call sum
  sw   a0, y         # y = sum()
  lw   ra, 0(sp)     # restore ra
  addi sp, sp, 4     # restore sp
  jr   ra            # return to OS
sum:
  add  a0, a0, a1    # a0 = a + b
  jr   ra            # return
```

Example Program: Symbol Table

Symbol	Address
`f`	0x10000000
`g`	0x10000004
`y`	0x10000008
`main`	0x00008000
`sum`	0x0000802C

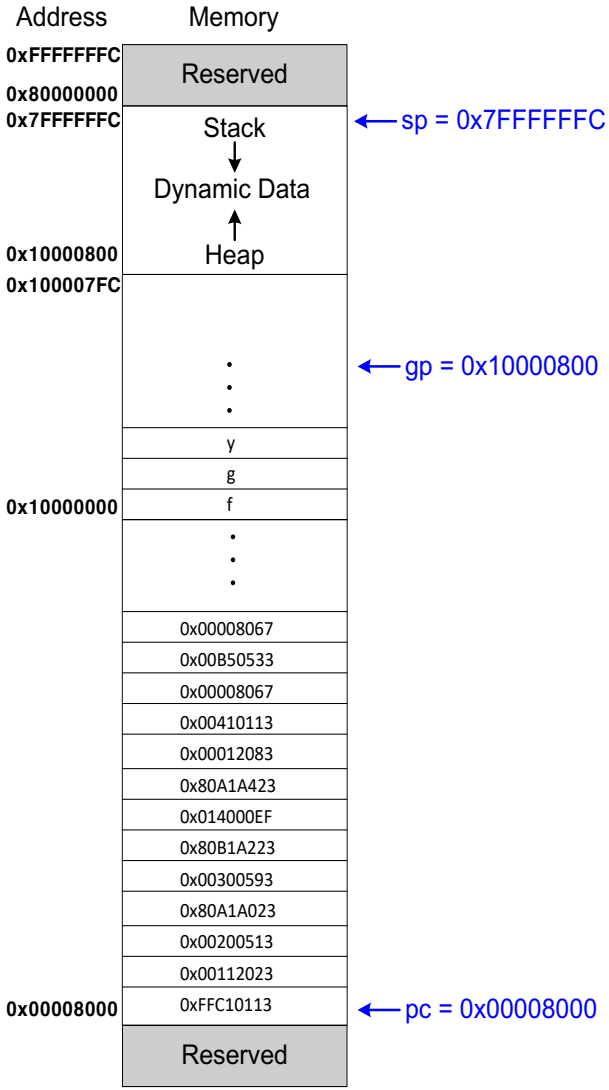
Example Program: Executable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction	
	0x00008000	0xFFC10113	addi sp, sp, -4
	0x00008004	0x00112023	sw ra, 0(sp)
	0x00008008	0x00200513	addi a0, zero, 2
	0x0000800C	0x80A1A023	sw a0, -2048(gp)
	0x00008010	0x00300593	addi a1, zero, 3
	0x00008014	0x80B1A223	sw a1, -2044(gp)
	0x00008018	0x014000EF	jal 0x0000802C
	0x0000801C	0x80A1A423	sw a0, -2040(gp)
	0x00008020	0x00012083	lw ra, 0(sp)
	0x00008024	0x00410113	addi sp, sp, -4
	0x00008028	0x00008067	jr ra
	0x0000802C	0x00B50533	add a0, a0, a1
	0x00008030	0x00008067	jr ra

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

Example Program: In Memory



Odds & Ends

- Pseudoinstructions
- Signed and unsigned instructions
- Floating point instructions

Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
`j label`	`jal zero, label`
`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`

See Appendix B, Table B.4 for more pseudoinstructions

Signed & Unsigned Instructions

- Multiplication and division
- Set less than
- Loads

Multiplication & Division

- **Signed:** `mulh`, `div`
- **Unsigned:** `mulhu`, `mulhsu`, `divu`

Set Less Than

- **Signed:** `slt`, `slti`
- **Unsigned:** `sltu`, `sltiu`

Note: `sltiu` sign-extends the immediate before comparing it to the register

Loads

- **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`

Addition & Subtraction

- RISC-V does not provide unsigned addition or instructions for detecting overflow because it can be done with existing instructions:
- Example: detecting unsigned overflow:

```
add  t0, t1, t2
bltu t0, t1, overflow
```

- Example: detecting signed overflow:

```
add  t0, t1, t2
slti t3, t2, 0          # t3=1 if t2 neg.
slt  t4, t0, t1         # t4=1 if result < t1
bne  t3, t4, overflow # overflow if:
                        # t2 neg & result>=t1 or
                        # t2 pos & result<t1
```

Floating Point Operations

- RISC-V offers three floating point extensions:
  - RVF: single-precision (32-bit)
  - RVD: double-precision (64-bit)
  - RVQ: quad-precision (128-bit)

Floating Point Registers

- **32** Floating point registers
- **Width** is highest precision – for example, if RVQ is implemented, registers are 128 bits wide
- When multiple floating point extensions are implemented, the lower-precision values occupy the lower bits of the register

Floating Point Registers

Name	Register Number	Usage
ft0-7	f0-7	Temporary variables
fs0-1	f8-9	Saved variables
fa0-1	f10-11	Function arguments/Return values
fa2-7	f12-17	Function arguments
fs2-11	f18-27	Saved variables
ft8-11	f28-31	Temporary variables

Floating Point Instructions

- Append .s (single), .d (double), .q (quad) for precision. I.e., `add.s`, `add.d`, and `add.q`
- **Arithmetic operations**:

  `fadd`, `fsub`, `fdiv`, `fsqrt`, `fmin`, `fmax`, multiply-add (`fmadd`, `fmsub`, `fnmadd`, `fnmsub`)
- **Other instructions:**

  move (`fmv.x.w`, `fmv.w.x`)

  convert (`fcvt.w.s`, `fcvt.s.w`, etc.)

  comparison (`feq`, `flt`, `fle`)

  classify (`fclass`)

  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)

Floating Point Instruction Formats

- Use R-, I-, and S-type formats
- Introduce another format for multiply-add instructions that have 4 register operands: R4-type