National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 2: ISA II

CS10014 Computer Organization

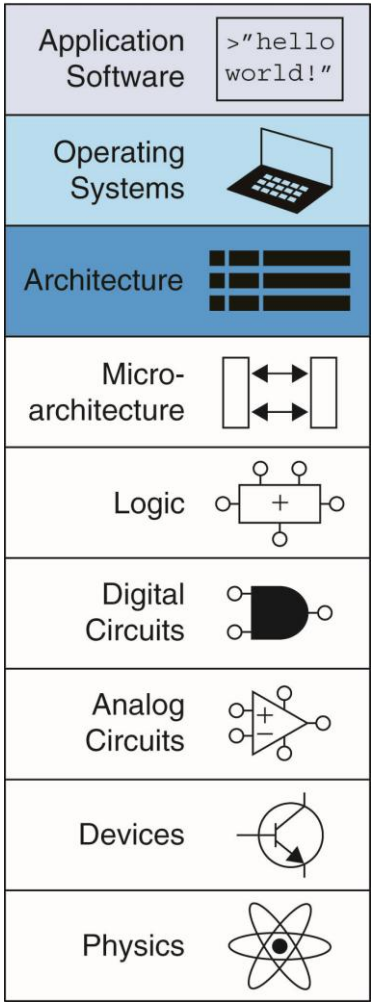
Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Outline

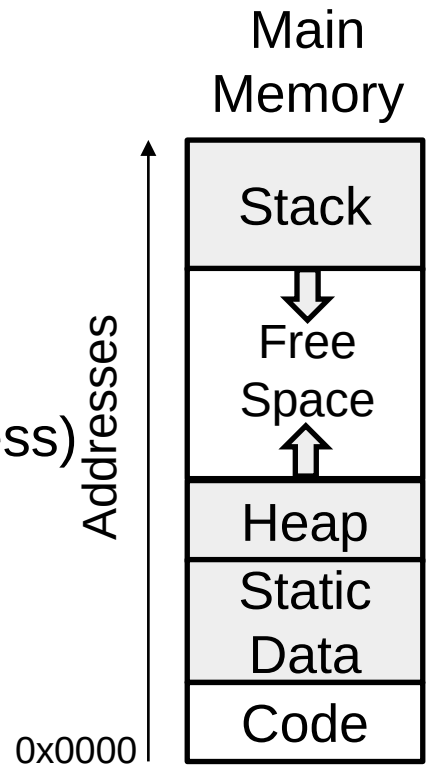
- Function call
- Recursive Function
- Machine Language
  - Instruction types and formats
  - Interpreting machine code
  - Addressing modes



Supporting Procedures in Computer Hardware

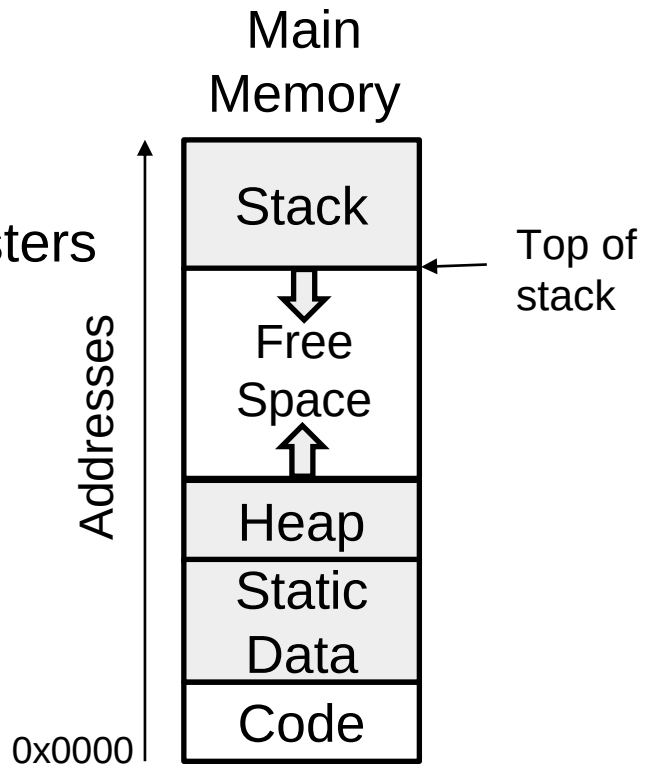
The Program Memory Layout

- A stored-program
  - Stores both data and code on memory
  - The code space is a memory space
    - stores program codes (the lowest address)
  - The static data space is a memory space
    - Store the program static data (global variables)
  - The heap space is a memory space
    - Managed by the memory allocation library (malloc())



The Program Memory Layout

- A stored-program
  - Stack
    - A data structure for spilling registers organized as a LIFO queue
  - The stack space is a memory space
    - Stores the program stack
    - Usually placed at the end (high addresses) of the memory



Program Stack

- Active routine is a routine (function)
  - Was invoked but didn't return yet
  - For example:
    - The routine fun is invoked by the bar routine, which also becomes active
    - The routine bar is invoked by the routine main
    - Initially, the main routine is active

```
int a = 10;

int main ()
{
      return  bar()  +  2;
}
int  bar()
{
      return  fun() +  4;
}
int  fun()
{
      return a;
}
```

Program Stack

- The set of active routines increases
  - Whenever a routine is invoked
- The set of active routines decreases
  - Whenever a routine returns
- The most natural data structure to keep track of active routines is a **stack**

Program Stack

- The program stack
  - A stack data structure that stores information belonging to active routines
    - Local variables, parameters, and return addresses
  - The program stack is stored in the main memory
  - Whenever a routine is invoked
    - Push the information belonging to the routine on the top of the stack, which causes it to grow
  - When a routine returns
    - Drop the contents at the top of the stack

Program Stack

- The stack pointer (sp)
  - A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found
  - In RISC-V, the stack pointer is stored by register sp or x2
- Return address
  - A link to the calling site that allows a procedure to return to the proper address
  - In RISC-V, the return address is stored in register x1

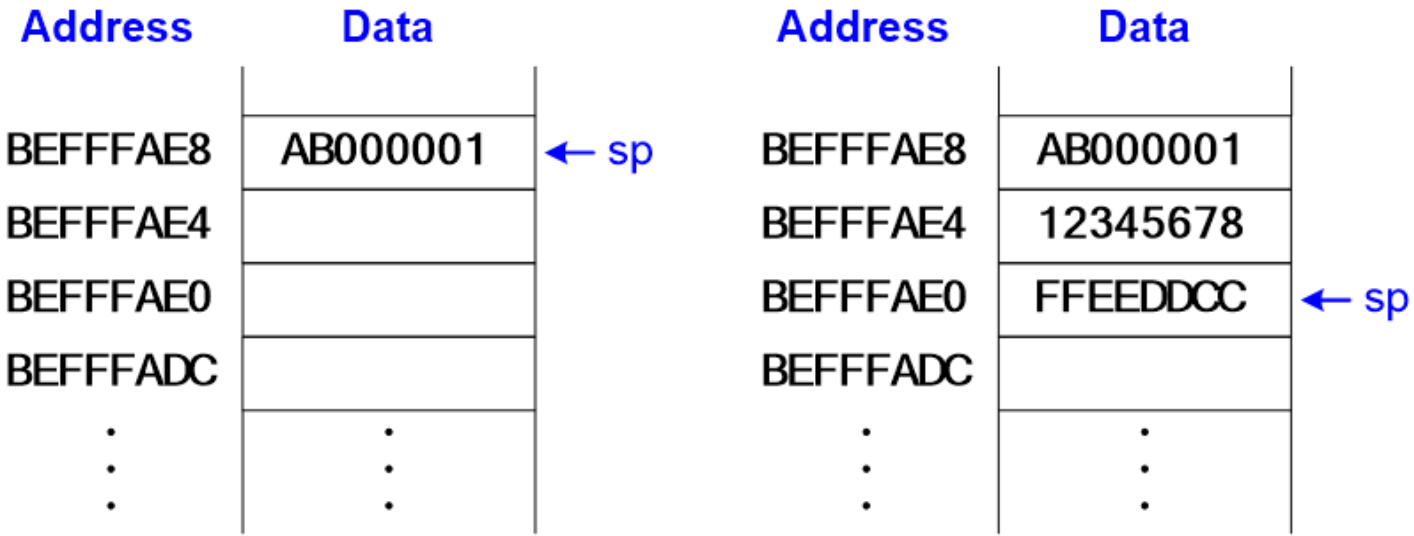
Program Stack

- The stack pointer
  - How to push the contents of register `a0` into stack
    - First, the stack pointer is decreased to allocate space (4 bytes)
    - The contents of register `a0` (4 bytes) are stored on the top of the program stack using the `sw` instruction

```
addi   sp, sp, -4    # allocate stack space
sw     a0, 0(sp)     # store data into stack
```

Program Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: sp points to top of the stack



Program Stack

- The stack pointer
  - How to pop a value from the top of the stack into register `a0`?
    - First, the value on the top of the program stack is loaded into register `a0` (4 bytes) using the `lw` instruction
    - Second, the stack pointer is increased to deallocate the space (4 bytes)

```
lw    a0, 0(sp)     # retrieve data from stack
addi  sp, sp, 4     # deallocate space
```

Function Calls

- Caller: calling function (in this case, main)
  - The program that instigates a procedure
  - Provide the necessary parameter values
- Callee: called function (in this case, sum)
  - A procedure that executes a series of stored instructions based on parameters provided by the caller
  - Then returns control to the caller

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

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

RISC-V Function Conventions

- Call Function: jump and link (jal)
- Return from function: jump register (jr ra)
- Arguments: a0 – a7
- Return value: a0

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

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  ra, simple  # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra         # return
```

void means that simple doesn't return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  ra, simple  # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra      # return
```

jal ra, simple:

ra = PC + 4 (0x00000304)

jumps to simple label (PC = 0x0000051c)

jr ra:

PC = ra (0x00000304)

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  simple      # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra          # return
```

- Preferred instruction:
  - jal simple – a pseudo-instruction for jal ra, simple
  - Pseudo-instructions are not actual RISC-V instructions but they are often simpler for the programmer
  - They are converted to real RISC-V instructions by the assembler

Returning Values from Routines

- Passing parameters
  - Parameter v is passed into register a0
  - How to invoke the pow2 routine to compute the square of 32

C Code

```
int pow2 (int v)
{
    return v*v;
}
```

RISC-V Assembly Code

```
Pow2:
  mul  a0,a0,a0 #a0:= a0 * a0
  ret           # return
```

RISC-V Assembly Code

```
main:
  li  a0,32 #set the parameter with value 32
  jal pow2  # invoke pow2
  ret       # return
```

Returning Values from Routines

- Reference parameters
  - A reference is a memory address
  - The information passed into or out of the routine must be located in the memory

C Code

```
int inc (int* v)
{
    *v = *v + 1;
}
```

RISC-V Assembly Code

```
inc:
  lw   a1,(a0)    #a1 := *v
  addi a1, a1, 1 #a1:= a1+1
  sw   a1, (a0)   #*v:= a1
  ret
```

RISC-V Assembly Code

```
.data
y: .skip 4
.text
Main:
  la  a0,y   #set the parameter with address of y
  jal inc   #a1:= a1+1
  ret
```

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments & Return Value

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2   # argument 0 = 2
addi a1, zero, 3   # argument 1 = 3
addi a2, zero, 4   # argument 2 = 4
addi a3, zero, 5   # argument 3 = 5
jal  diffofsums    # call function
add  s7, a0, zero  # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1    # t0 = f + g
add  t1, a2, a3    # t1 = h + i
sub  s3, t0, t1    # result = (f + g) - (h + i)
add  a0, s3, zero  # put return value in a0
jr   ra            # return to caller
```

`jal diffofsums`
is pseudocode for
`jal ra, diffofsums`

Input Arguments & Return Value

RISC-V assembly code

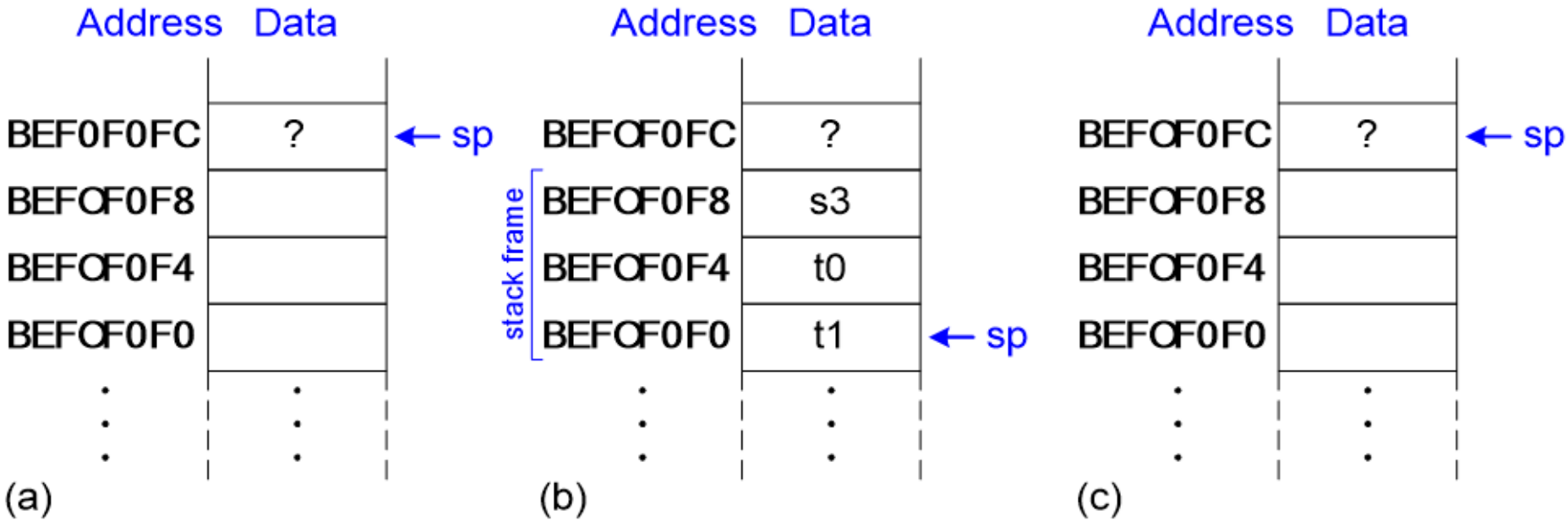
```
# s3 = result
diffofsums:
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  s3, t0, t1   # result = (f + g) - (h + i)
  add  a0, s3, zero # put return value in a0
  jr   ra           # return to caller
```

- diffofsums overwrote 3 registers: t0, t1, s3
- diffofsums can use the stack to temporarily store registers

Storing Register Values on the Stack

```
# s3 = result
diffofsums:
    addi sp, sp, -12        # make space on stack to
                            # store three registers
    sw   s3, 8(sp)          # save s3 on stack
    sw   t0, 4(sp)          # save t0 on stack
    sw   t1, 0(sp)          # save t1 on stack
    add  t0, a0, a1         # t0 = f + g
    add  t1, a2, a3         # t1 = h + i
    sub  s3, t0, t1         # result = (f + g) - (h + i)
    add  a0, s3, zero       # put return value in a0
    lw   t1, 0(sp)          # restore $t1 from stack
    lw   t0, 4(sp)          # restore $t0 from stack
    lw   s3, 8(sp)          # restore $s3 from stack
    addi sp, sp, 12         # deallocate stack space
    jr   ra                 # return to caller
```

The Stack During diffofsums Call



Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
   addi sp, sp, -4          # make space on stack to
                            # store one register
   sw   s3, 0(sp)           # save s3 on stack
   add  t0, a0, a1          # t0 = f + g
   add  t1, a2, a3          # t1 = h + i
   sub  s3, t0, t1          # result = (f + g) - (h + i)
   add  a0, s3, zero        # put return value in a0
   lw   s3, 0(sp)           # restore $s3 from stack
   addi sp, sp, 4           # deallocate stack space
   jr   ra                  # return to caller
```

Optimized diffofsums

Save the usage of the s0 register, and reuse a0 register

```
# a0 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  a0, t0, t1    # result = (f + g) - (h + i)
  jr   ra            # return to caller
```

Non-Leaf Function Calls

Non-leaf function:

a function that calls another function

```
func1:
  addi sp, sp, -4     # make space on stack
  sw   ra, 0(sp)      # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)      # restore ra from stack
  addi sp, sp, 4      # deallocate stack space
  jr   ra             # return to caller
```

Save the return address (ra or x1 register) to the stack before making a nested call -> the original return location is not overwritten

Non-Leaf Function Calls

- Key steps for a non-leaf function calls
  - Function Prologue
    - At the beginning of the non-leaf function, a stack must be set up to preserve necessary registers
    - Saving the current return address (ra)
    - Saving any callee-saved registers (s0-s11)
    - Saving its own arguments registers (a0-a7)
  - Function Body

Non-Leaf Function Calls

- Key steps for a non-leaf function calls
  - Function Epilogue (Exit)
    - Restoring saved registers
    - Incrementing the stack pointer (sp)
    - Returning to the original caller using the “jr ra” or “ret” instruction

Recursive Functions

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

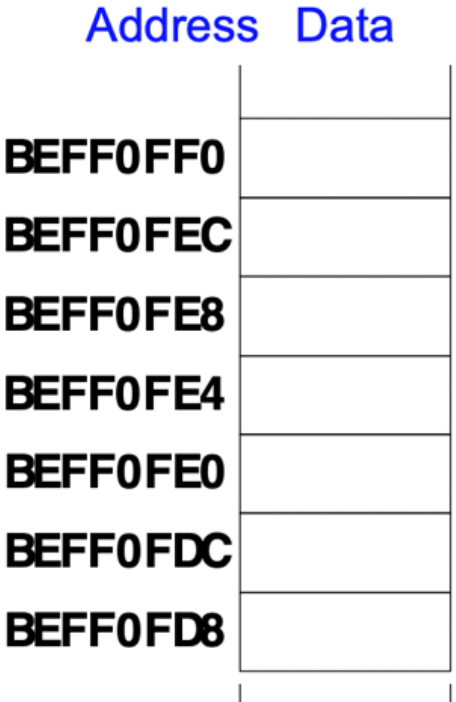
Recursive Functions

```
0x8500 factorial: addi sp, sp, -8     # make room for a0, ra
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1    # temporary = 1
0x8510            bgt  a0, t0, else   # if n>1, go to else
0x8514            addi a0, zero, 1    # otherwise, return 1
0x8518            addi sp, sp, 8      # restore sp
0x851C            jr   ra             # return
0x8520 else:      addi a0, a0, -1     # n = n - 1
0x8524            jal  factorial      # recursive call
0x8528            lw   ra, 0(sp)      # restore ra
0x852C            lw   t1, 4(sp)      # restore n into t1
0x8530            addi sp, sp, 8      # restore sp
0x8534            mul  a0, t1, a0     # a0 = n*factorial(n-1)
0x8538            jr   ra             # return
```

Recursive Functions

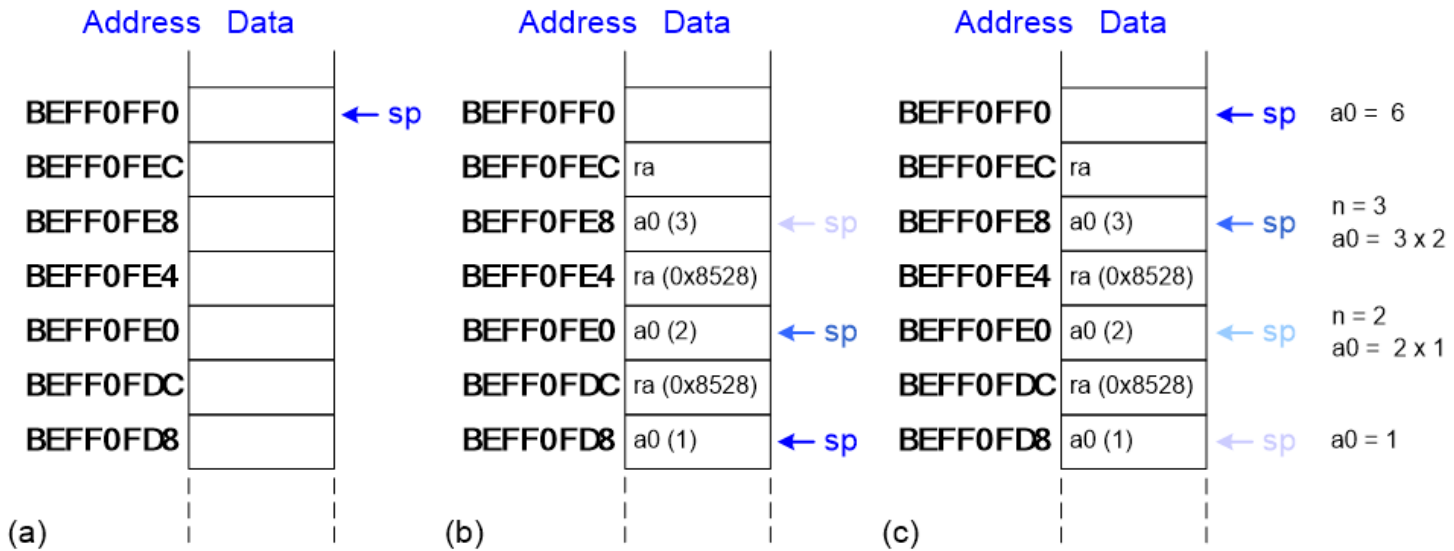
What does the stack look like when executing `factorial(3)`?

```
0x8500 factorial: addi sp, sp, -8
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1
0x8510            bgt  a0, t0, else
0x8514            addi a0, zero, 1
0x8518            addi sp, sp, 8
0x851C            jr   ra
0x8520 else:      addi a0, a0, -1
0x8524            jal  factorial
0x8528            lw   ra, 0(sp)
0x852C            lw   t1, 4(sp)
0x8530            addi sp, sp, 8
0x8534            mul  a0, t1, a0
0x8538            jr   ra
```



Recursive Functions

Stack (a) before, (b) during, and (c) after recursive call.



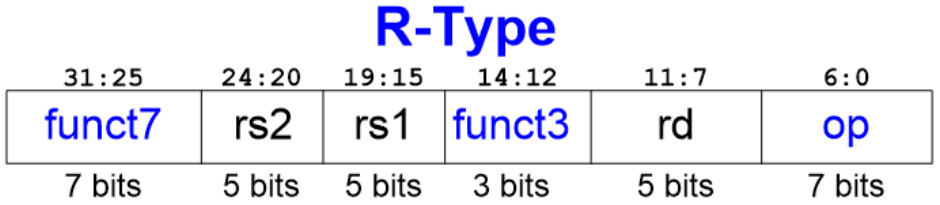
Representing Instructions in the Computer

Machine Language

- Binary representation of instructions
- Computers only understand 1’s and 0’s
- **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, func3:
    - The function (7 bits and 3-bits, respectively)
    - With opcode, tells computer what operation to perform



R-Type

Assembly / **Field Values**

Assembly	funct7	rs2	rs1	funct3	rd	op	
`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
```

R-Type

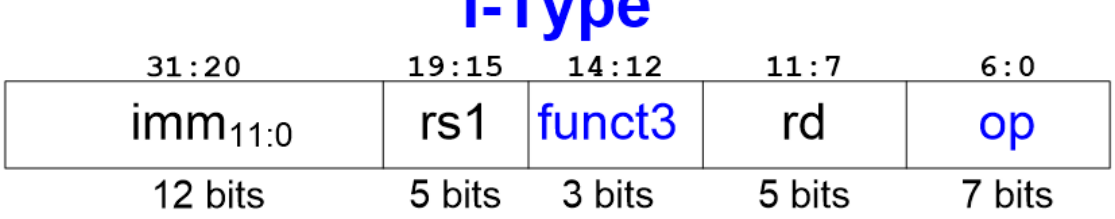
Assembly | **Field Values**

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`
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

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	

I-Type



- Immediate-type
- 3 register operands:
  - rs1: register source operand
  - rd: register destination operand
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the operation code or opcode
  - Funct3:
    - The function (3-bit function code)
    - With opcode, tells computer what operation to perform

I-Type

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

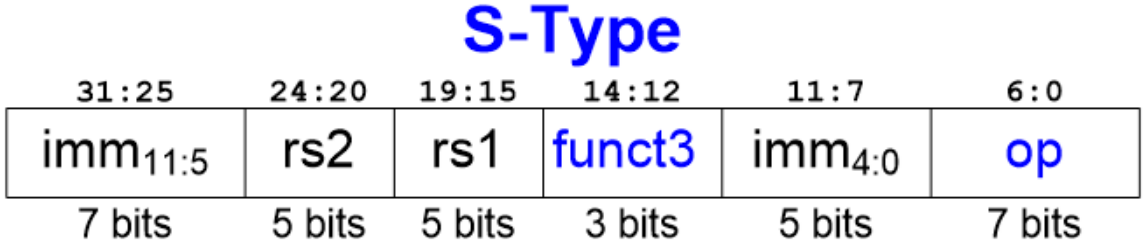
$imm_{11:0}$	rs1	funct3	rd	op	
0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
0000 0001 1011	00000	001	01001	000 0011	(0x01B01483)
0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
12 bits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

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

`addi rd, rs1, imm`

`lw rd, imm(rs1)`

S-Type



- Store-type
- 3 register operands
  - rs1: base register
  - rs2: value to be stored to memory
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the operation code or opcode
  - Funct3:
    - The function (3-bit function code)
    - With opcode, tells computer what operation to perform

S-Type

Assembly

```
sw t2, -6(s3)
sh s4, 23(t0)
sb t5, 0x2D(zero)
```

Field Values

	$imm_{11:5}$	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	

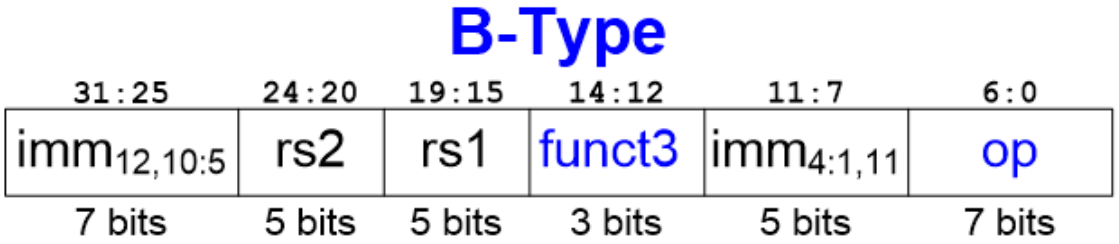
Machine Code

	$imm_{11:5}$	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	

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

`sw rs2, imm(rs1)`

B-Type



- Branch-type
- 3 register operands:
  - rs1: register source 1
  - rs2: register source 2
  - imm: 12-bit two's complement immediate - address offset
- Other fields:
  - op: the operation code or opcode
  - Funct3:
    - The function (3-bit function code)
    - With opcode, tells computer what operation to perform

B-Type

- The 12-bit immediate encodes where to branch (relative to the branch instruction)
  - 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	X	

B-Type

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)
```

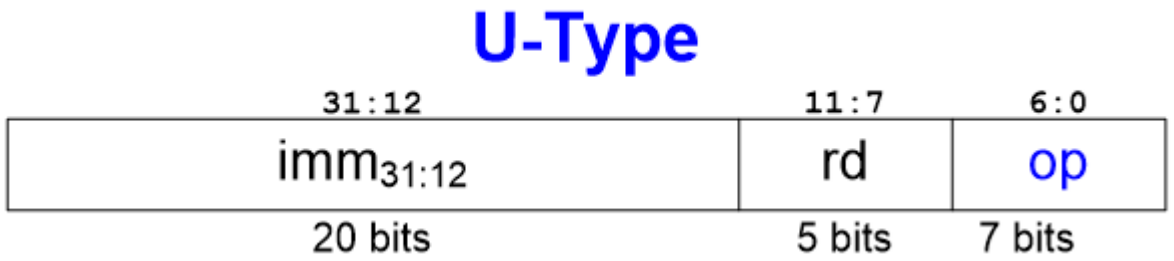
Note the differing order of operands in assembly and machine codes:

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

$\texttt{beq rs1, rs2, imm}_{12:1}$

U-Type

- Upper-immediate Type
- Used for load upper immediate (lui)
- 2 operands
  - rd: destination register
  - $Imm_{31:12}$: upper 20 bits of 1 32-bit immediate
- Other fields:
  - op: the operation code or opcode – tells computer what operation to perform

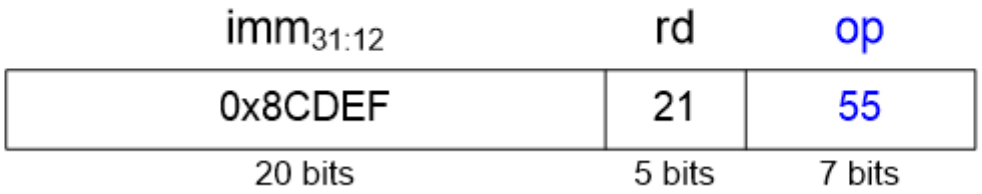


U-Type

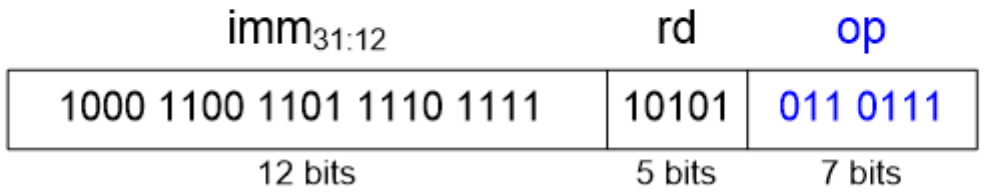
Assembly

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

Field Values



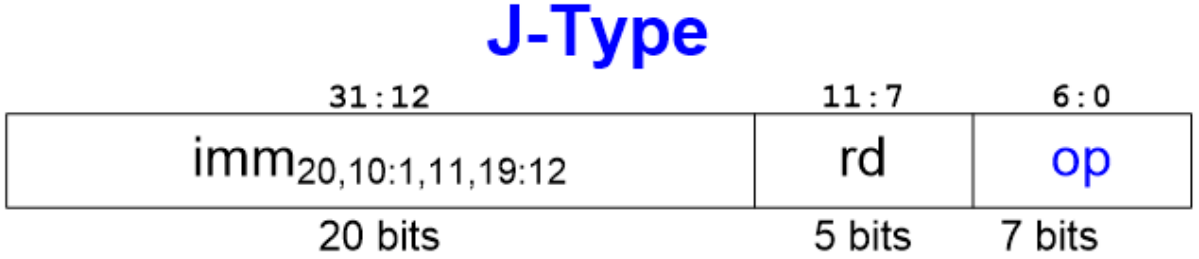
Machine Code



(0x8CDEFAB7)

J-Type

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



J-Type

- **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	X
bit number	20		19	18	17	16		15	14	13	12		11	10	9	8		7	6	5	4		3	2	1	

J-Type

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

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	

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
```

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												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

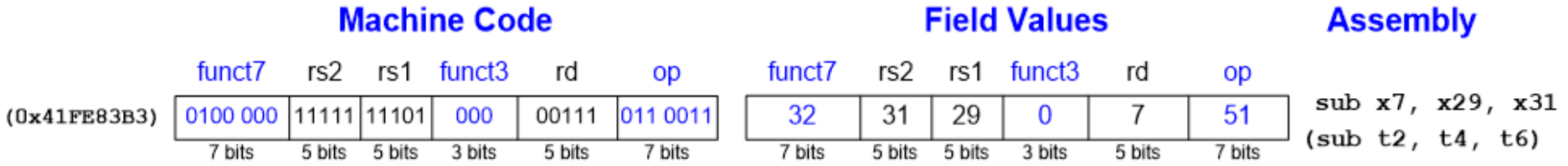
op = 51: R-type

0xFDA48393: 1111 1101 1010 0100 1000 0011 1001 0011

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

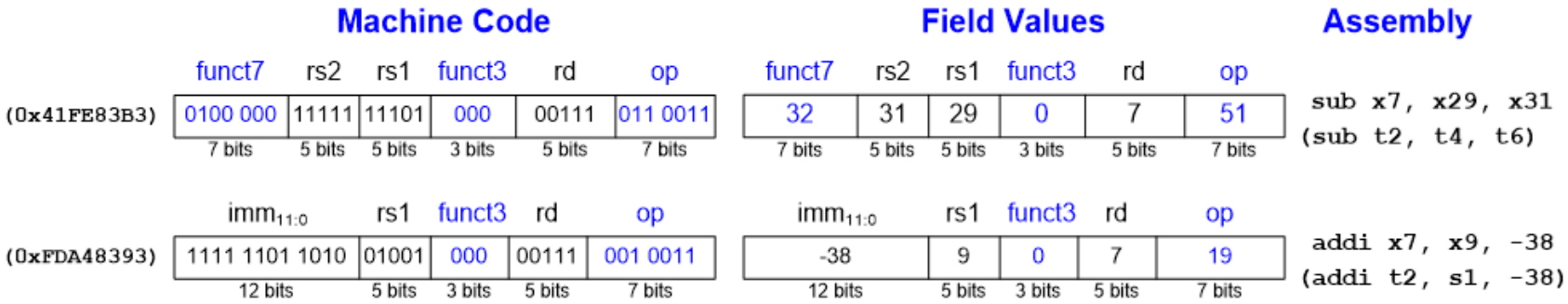
Interpreting Machine Code

- 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



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



RISC-V: Addressing Modes

- How do we address the operands?
  - Register only
  - Immediate
  - Base addressing
  - PC-relative

Register Only

- Operands found in registers
  - **Example:** add s0, t2, t3
  - **Example:** sub t6, s1, 0

Immediate

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

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

2908/4 = 727

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	

Generating Constants

- 12-bit signed constants using addi:

C Code

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

RISC-V assembly code

```
# s0 = a
addi s0, 0, -372
```

- Any immediate that needs more than 12 bits cannot use this method

Generating 32-bit Constants

- Use load upper immediate (**lui**) and **addi**:
  - **lui**: puts an immediate in the upper 20 bits of destination register, 0's in lower 12 bits

C Code

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

RISC-V assembly code

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

  - Remember that addi sign-extends its 12-bit immediate

Generating 32-bit Constants

- If bit 11 of 32-bit constant is 1, increment upper 20 bits by 1 in lui
  - if the MSB of the 12-bit constant (i.e. bit 11) is a 1, the constant is then sign extended.

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
```

0xFFFFF = -1

Signed extension

-341 = 0xEAB =
1110 1010 1011

bit 11 of 32-bit

RISC-V: Pseudo-instruction

- Load immediate 32-bit word is tedious
- Pseudo-instruction
  - Assembler program translate “Load immediate” instruction “li” to two real RISC-V instructions: “lui” and “addi’

C Code

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

Note: -341 = 0xEAB

RISC-V pseudoinstructions

```
# s0 = a
li   s0, 0xFEDC8EAB
```

RISC-V real instructions

```
# s0 = a
lui  s0, 0xFEDC9
addi s0, s0, 0xEAB
```

RISC-V: Pseudo-instruction

- There is no instruction to load a register with a constant value
  - To load s0 with the small constant 6, we use the instruction
    - addi s0, zero, 6
  - To load s0 with a large constant 0xFEDC8EAB
    - lui s0, 0xFEDC9
    - addi s0, s0, 0xEAB
  - To load a register with a constant of any size constant (up to 32 bits)
    - li s0, 6
    - li s0, 0xFEDC9

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

The first step is to load array[i] into a temporary register. Before we can load array[i] into a temporary register, we need to have its address.

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,0       // i = 0
loop1:
   slli x6,x5, 2   // x6 = i * 4
   add  x7,x10,x6  // x7 = address
                   // of array[i]
   SW   x0,0(x7)   // array[i] = 0
   addi x5,x5,1    // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

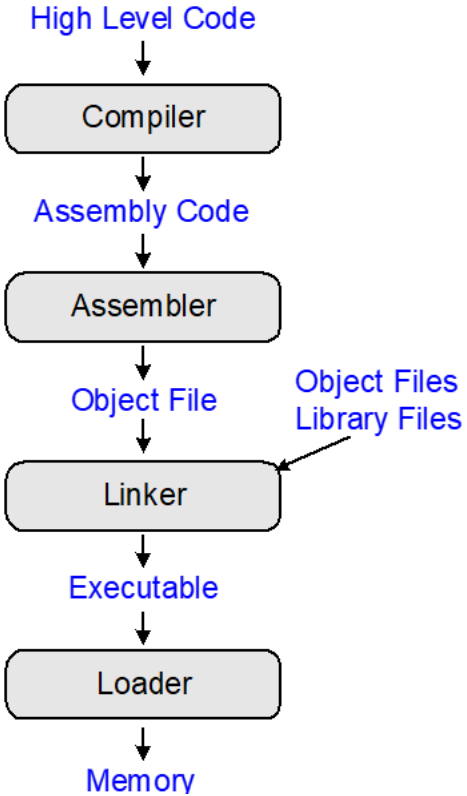
X10 is the base address of array

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   mv x5,x10       // p = address
                   // of array[0]
   slli x6,x11, 2  / x6 = size * 4
   add x7,x10,x6   // x7 = address
                   // of array[size]
loop2:
  SW x0,0(x5)      // Memory[p] = 0
   addi x5,x5, 4   // p = p + 4
   bltu x5,x7,loop2
                   // if (p<&array[size])
                   // go to loop2
```

mv rd, rs

How to Compile & Run a Program



The Compilation Pipeline

- How source code becomes a running program ?
  - **Preprocessing**
    - The preprocessor handles #include and #define statement
  - **Compilation**
    - The compiler turns C/C++ code into assembly language
  - **Assembly**
    - The assembler converts assembly into object files (.o or .obj)
  - **Linking**
    - The linker combines object files and libraries into an executable

Linker

- **Input**
  - Object code files, information tables (e.g. foo.o, lib.o for RISC-V)
- **Output**
  - Executable code (e.g. a.out for RISC-V)
- Combines several object (.o) files into a single executable (“linking”)
- Enables separate compilation of files
  - The linker’s job is to figure out where every reference and definitions should live in the final executable

What are symbols?

- **A symbol**
  - Any name entity in your program that the linker needs to keep track of. The symbol includes
  - **Functions** (main, printf, malloc)
  - **Global variables** (extern int counter)
  - **Static variables** (those marked with static)
  - **Class methods**
    - Ex. When you write int global_counter = 42; at the top of your C file, you create a symbol called global_counter that points to a specific spot in memory where the value 42 lives

Symbol Types

- The linker categorizes symbols into two types that affect how conflicts are resolved
- **Strong symbols includes**
  - Functions that are defined (have a body)
  - Initialized global variables
- **Weak symbols**
  - Uninitialized global variables
  - Function declarations without definitions

Symbol Table

- The symbol table: A linker's phonebook
  - Every object file contains a symbol table

Symbol Table Structure

Symbol Name	Address	Size	Type	Binding	Section	Description
main	0x08048400	24	FUNC	GLOBAL	.text	Main function entry point
printf	undefined	-	FUNC	GLOBAL	UND	External library function
global_counter	0x08049540	4	OBJECT	GLOBAL	.data	Initialized global variable
static_var	0x08049544	4	OBJECT	LOCAL	.data	Static variable (file scope)
uninitialized_array	0x08049580	1024	OBJECT	WEAK	.bss	Uninitialized global array
helper_function	0x08048450	16	FUNC	GLOBAL	.text	User-defined function

Symbol Table Legend

Type
- FUNC - Function
- OBJECT - Variable
- SECTION - Section

Binding
- GLOBAL - Visible globally
- LOCAL - File scope only
- WEAK - Can be overridden

Section
- .text - Executable code
- .data - Initialized data
- .bss - Uninitialized data

Address
- 0x08048xxx - Code segment
- 0x08049xxx - Data segment
- undefined - External ref

https://reurl.cc/j6M1zq

Symbol Table

- We can peek at the symbol table using tools
  - “nm” on Unix or “dumpbin” on Windows

```
# Compile a simple C file
gcc -c example.c -o example.o

# Look at the symbol table
nm example.o
```

```
0000000000000000 T main
                 U printf
0000000000000004 D global_var
```

https://reurl.cc/j6M1zq

Symbol Table Example

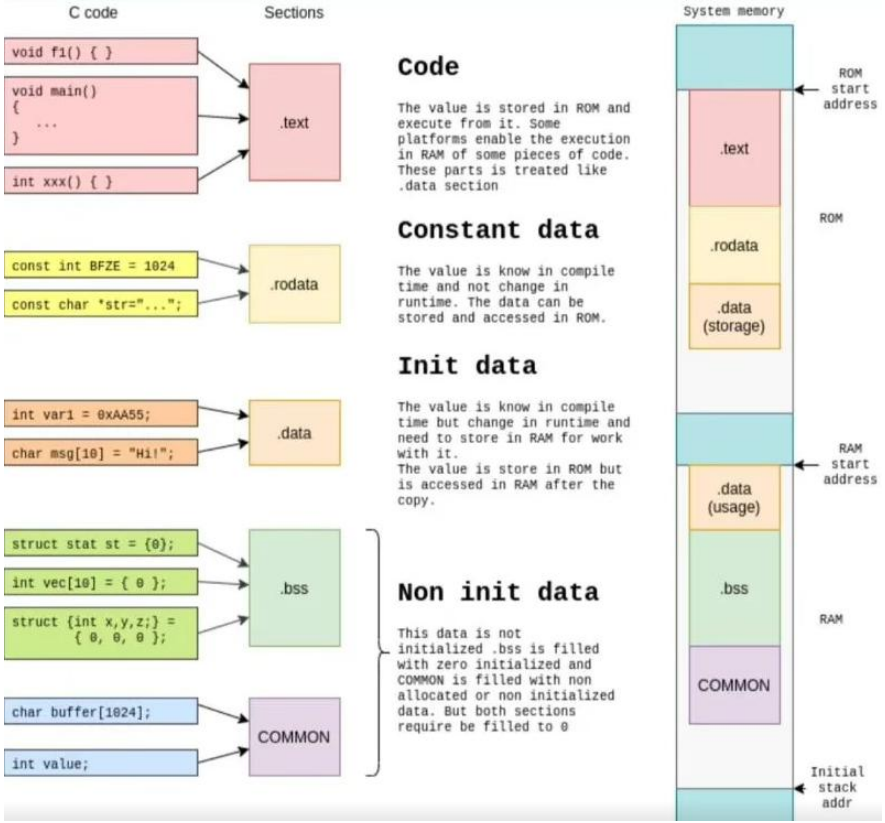
Symbol	Type
bar	U
dec	U
main	T
“Here”	D
num	D
printf	U
“%\n”	D

U: undefined indicates the external file reference
T: .Text section
D: .Data section

```
1  #include <stdio.h>
2  extern int bar();
3  extern int dec;
4  int main() {
5       char *output = “Here”;
6       static int num = 7;
7       int i = 5;
8       while (i > 0) {
9            i --;
10           int temp = bar(num);
11           printf(“%d\n”, temp);
12      }
13}
```

Where symbols live: The ELF Format

- On Unix-like systems, object files and executables use **ELF (Executable and Linkable Format) format**



https://reurl.cc/j6M1zq

A ELF File

- A ELF file is divided into sections
  - **.text**: The actual machine code
  - **.data**: Initialized global variables
  - **.bss**: Uninitialized global variables
  - **.symtab**: The symbol table
  - **.rel.text and .rel.data**: Relocation entries

https://reurl.cc/j6M1zq

Relocation

- Symbol resolution
  - Tell the linker which symbols go with which definitions
- Relocation
  - This is where the linker patches up all the addresses in your code

```
# Before linking
call printf    # This is actually "call <placeholder>"
mov eax, [global_var]  # This is "mov eax, [<placeholder>]"

# After linking
call 0x08048370    # Actual address of printf
mov eax, [0x08049540]  # Actual address of global_var
```

https://reurl.cc/j6M1zq

Shared Libraries and Dynamic Loading

- **Static linking**
  - Create static executables where everything is bundled together
- **With shared libraries**
  - .so file on Unix, .dll files on Windows
  - Some symbols aren’t resolved until the program actually runs
- **The dynamic linker**
  - ld.so on Linux handles this runtime symbol resolution

https://reurl.cc/j6M1zq

Shared Libraries and Dynamic Loading

- **The dynamic linker**
  - ld.so on Linux handles this runtime symbol resolution
  - **Lazy binding:**
    - Function addresses are resolved only when first called
  - **Global symbol interposition**
    - Symbols in the main program can override library symbols

https://reurl.cc/j6M1zq

Conclusion

- Function call
- Recursive Function
- Machine Language
  - Instruction types and formats
  - Interpreting machine code
  - Addressing modes