Architecture and Assembly Programming

University of South Carolina

Introduction to Computer Architecture
Fall, 2024

Mehdi Yaghouti

Molinaroli College of Engineering and Computing
UNIVERSITY OF SOUTH CAROLINA

Architecture

- The architecture is the programmer's view of a CPU
- The architecture is mainly defined by Instruction set and the Operand locations

Architecture

- The architecture is the programmer's view of a CPU
- The architecture is mainly defined by Instruction set and the Operand locations
- Each instruction includes both the operation to perform and the necessary operands

Architecture

- The architecture is the programmer's view of a CPU
- The architecture is mainly defined by Instruction set and the Operand locations
- Each instruction includes both the operation to perform and the necessary operands
- Instructions will be eventually encoded in `machine language` as 0s and 1s
- It is way easier to write symbolic codes in `assembly language` rather than 0s and 1s
- Each CPU has its own specific instructions set

Architecture

- The architecture is the programmer's view of a CPU
- The architecture is mainly defined by Instruction set and the Operand locations
- Each instruction includes both the operation to perform and the necessary operands
- Instructions will be eventually encoded in `machine language` as 0s and 1s
- It is way easier to write symbolic codes in `assembly language` rather than 0s and 1s
- Each CPU has its own specific instructions set
- The instruction sets of different architectures have a lot in common
- Once one instruction set is learned, understanding others is fairly straightforward
- Here we are more concerned about the common concepts rather than the differences

Architecture

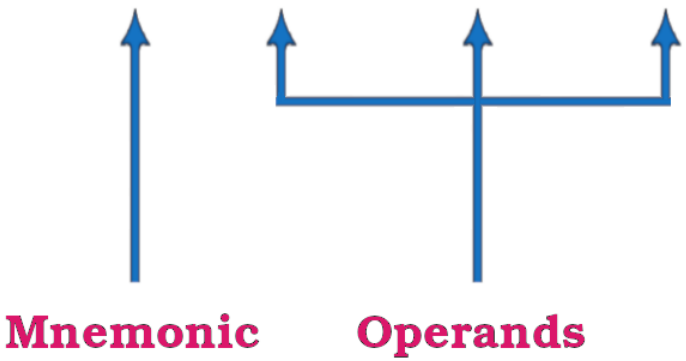
- The architecture is the programmer's view of a CPU
- The architecture is mainly defined by Instruction set and the Operand locations
- Each instruction includes both the operation to perform and the necessary operands
- Instructions will be eventually encoded in `machine language` as 0s and 1s
- It is way easier to write symbolic codes in `assembly language` rather than 0s and 1s
- Each CPU has its own specific instructions set
- The instruction sets of different architectures have a lot in common
- Once one instruction set is learned, understanding others is fairly straightforward
- Here we are more concerned about the common concepts rather than the differences
- In this course we focus on RISC-V architecture
- RISC-V is the first open source architecture with broad commercial support
- The architecture does not define the underlying hardware
- One architecture might have several implementations

Assembly language

- Assembly language is the human-readable representation of machine language
- Defined by Instructions and operand locations
- The operand determines the physical location from which the data must be retrieved
- **RISC-V Operands:**
  - Immediate Values
  - Registers
  - Memory

Instruction Structure

```
add x5, x6, x7        # x5 ← x6 + x7
```



- **Mnemonic:** A symbolic abbreviation used to represent an instruction
- **Operands:** Are the values or data on which the instruction operates

Operands

- Different operand types provides a range of trade-off between speed and capacity
- **Constants**:
  - Values directly embedded in the instruction itself
- **Registers**:
  - Fast storage locations directly accessible by the CPU
  - Store small amounts of data
- **Memory**:
  - Larger storage space, but slower access compared to registers
  - Used for additional data beyond what can be stored in registers
- An architecture is called 32-bits when it operates on 32-bit data
- Here we focus on 32-bit version of RISC-V (RV32I)

R-Type Instructions

- Some instructions have three operands

```
add x5, x2, x3      # x5 ← x2 + x3
sub x4, x6, x7      # x4 ← x6 - x7
and x5, x6, x8      # x5 ← x6 & x8
```

- We call this instructions R-Type
- The registers are ordered as destination, source-1 and source-2

Registers

- Commonly used operands are kept in registers for faster access
- RISC-V Registers set

Name	Register Number	Use
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	Temporary registers
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	Temporary registers

I-Type Instructions

- Some instructions have two register operands and one immediate operand

```
addi x3, x6, 4          # x3 ← x6 + 4
andi x5, x4, 0xA        # x5 ← x4 & 0xA
slli x5, x6, 3          # x5 ← x6 << 3
```

- We call this instructions I-Type
- In these instructions the last operand is a 12-bit immediate value
- The immediate value can be in given in decimal, hexadecimal or binary formats

I-Type Instructions

- Some instructions have two register operands and one immediate operand

```
addi x3, x6, 4          # x3 ← x6 + 4
andi x5, x4, 0xA        # x5 ← x4 & 0xA
slli x5, x6, 3          # x5 ← x6 << 3
```

- We call this instructions I-Type
- In these instructions the last operand is a 12-bit immediate value
- The immediate value can be in given in decimal, hexadecimal or binary formats

```
addi x5, x2, 0xA
addi x5, x7, 0b1010
```

Immediates

- **Immediate Values**:
  - Directly available from the instruction itself
  - Do not require access to registers or memory
- **Format**:
  - Immediates can be written in decimal, hexadecimal, or binary
  - **Hexadecimal** constants start with `0x`
  - **Binary** constants start with `0b`, similar to C
- **Representation**:
  - Immediates are 12-bit two's complement numbers
  - They are sign-extended to 32 bits
- **Common Usage**:
  - The `addi` is used to assign a constant to a register
  - Example Code:

```
addi s5,x0,0b1101101
addi s5,x0,0x6D
addi s5,x0,109
```

Example

- Adding two decimal values

Assembly Code

```
addi x1, zero, 3      # x1<-- 3
addi x2, zero, 4      # x2<-- 4
add  x3, x1, x2       # x3<--  x1 + x2
```

Example

- Adding two hexadecimal values

Assembly Code

```
addi x1, zero,0xA23     # x1<--  0xA23
addi x2, zero,0x1B0     # x2<--  0x1B0
add  x3, x1, x2         # x3<--   x1 + x2
```

Example

- Registers as local variables

High-Level Code

```
int b=4, c=3;
int g=-1, h=-2, i=1, j=2;

a = b - c
f = (g+h) - (i+j)
```

Assembly Code

```
addi s1, zero, 4
addi s2, zero, 3
addi s4, zero, -1
addi s5, zero, 0xFFE
addi s6, zero, 1
addi s7, zero, 0b10
sub s0, s1, s2     #  a = b - c
add t0, s4, s5     # t0 = g +h
add t1, s6, s7     # t1 = i + j
sub s3, t0, t1     # f  = (g+h) - (i+j)
```

lui

- `lui`, loads 20-bit immediate into the upper part of register

```
lui s0, 0xABCDE     # s0 ← ABCDE000
```

lui

- `lui`, loads 20-bit immediate into the upper part of register

```
lui s0, 0xABCDE     # s0 ← ABCDE000
```

- Loading constants into registers

High-Level Code	Assembly Code
int a=0xABCDE123;	lui s0, 0xABCDE #s0 = 0xABCDE000 addi s0, s0, 0x123 #s0 = 0xABCDE123

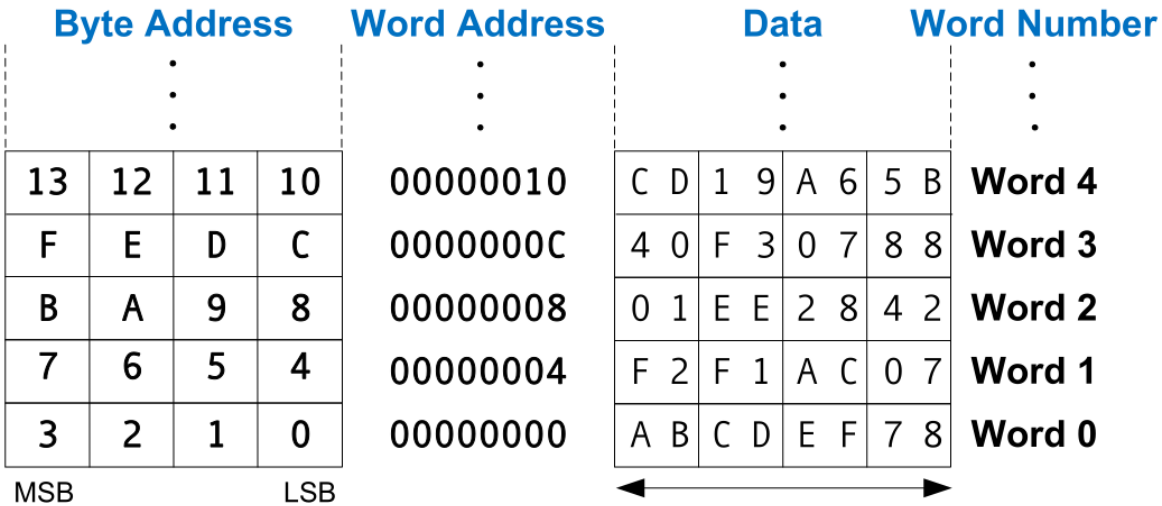
Example

- Loading constants into registers

High-Level Code	Assembly Code	
int a=0xABCDE123;	lui s0, 0xABCDE	#s0 = 0xABCDE000
int b=0xFEEDA787;	addi s0, s0, 0x123	#s0 = 0xABCDE123
int c=a & b;	lui s1, 0xFEEDA	#s1= 0xFEEDA000
	addi s1, s1, 0x787	#s1= 0xFEEDA787
	and s2, s1, s0	#s2= s1 & s0

Memory

- **Registers and Memory**:
  - Registers are small and fast, but limited to 32 variables
  - Memory provides larger storage but is slower
  - Frequently used variables are stored in registers for quick access
- **RISC-V Architecture**
  - Instructions operate exclusively on registers
  - Data in memory must be moved to registers before processing
  - The architecture we focus on, uses 32-bit memory addresses and data words
- **Byte-Addressable Memory**:
  - We take memory as byte-addressable, where each byte has a unique address



Memory Access

- Only `Load` and `Store` instructions access the memory
- These two instructions use base addressing mode
- **Base addressing mode**
  1. Base address is stored in a register
  2. Effective memory address is formed by adding an immediate offset to the base
- Load Example

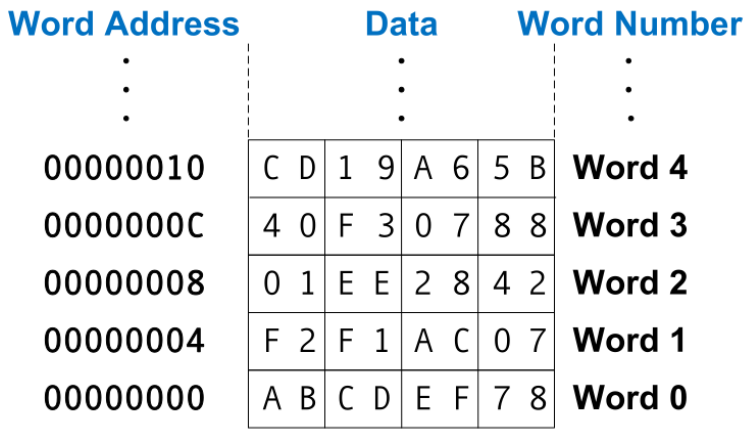
```
addi x6, x0, 0      # x6 ← 0
lw x5, 12(x6)       # x5 ← Mem[x6+12]
```

Load Word

- Load Example

```
addi x6, x0, 0      # x6 ← 0
lw x5, 12(x6)       # x5 ← Mem[x6+12]
```

x5 ← 0X40F30788

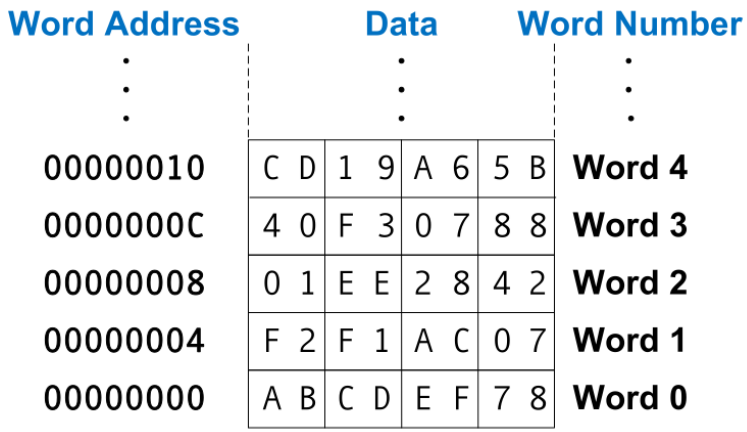


Load Half-Word

- Load the sign-extended lower half of the word

```
addi x6, x0, 0      # x6 ← 0
lh x5, 12(x6)       # x5 ← Mem[x6+12]
```

x5 ← 0X00000788

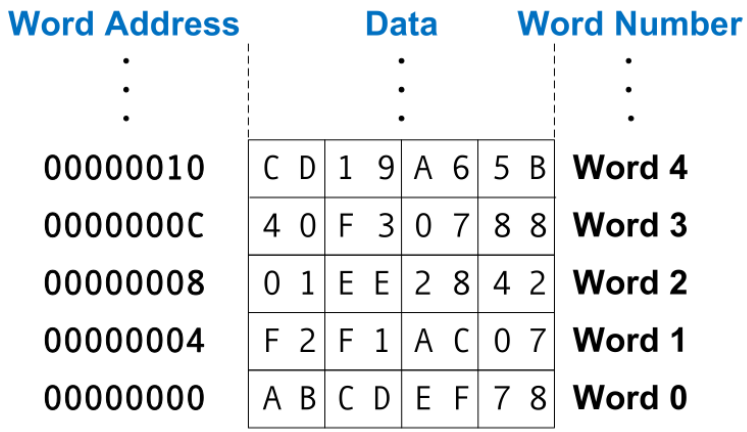


Load Byte

- Load the sign-extended byte

```
addi x6, x0, 0      # x6 ← 0
lb x5, 12(x6)       # x5 ← Mem[x6+12]
```

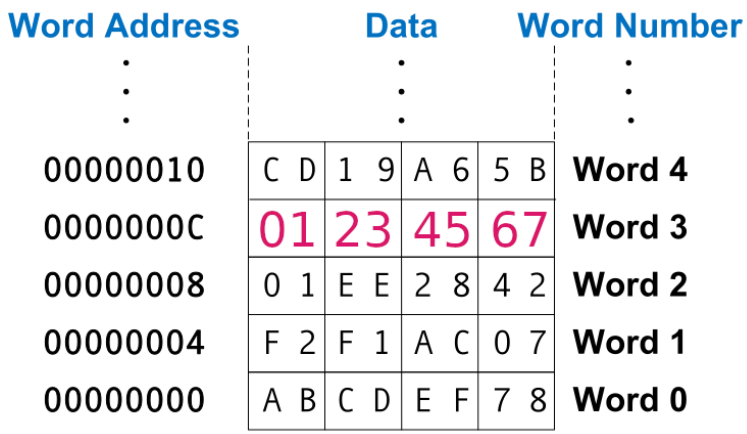
x5 ← 0XFFFFFF88



Save Word

- Save Word example

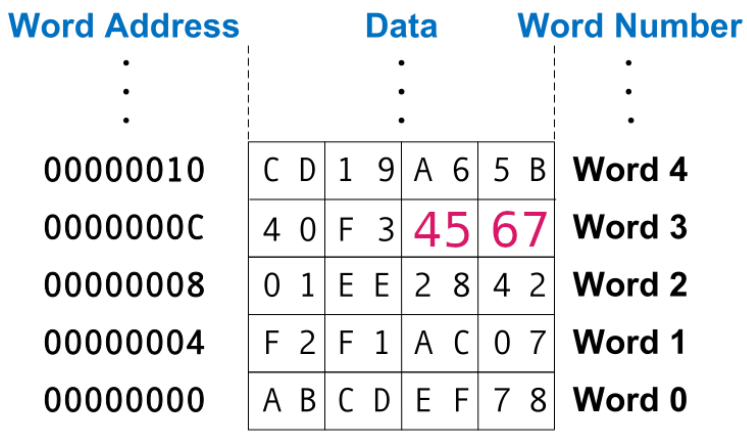
```
addi x6, x0, 0
lui x5, 0x01234
ori x5, x5, 0x567
sw x5, 12(x6)        # x5 → Mem[x6+12]
```



Save Half-Word

- Save half-word example

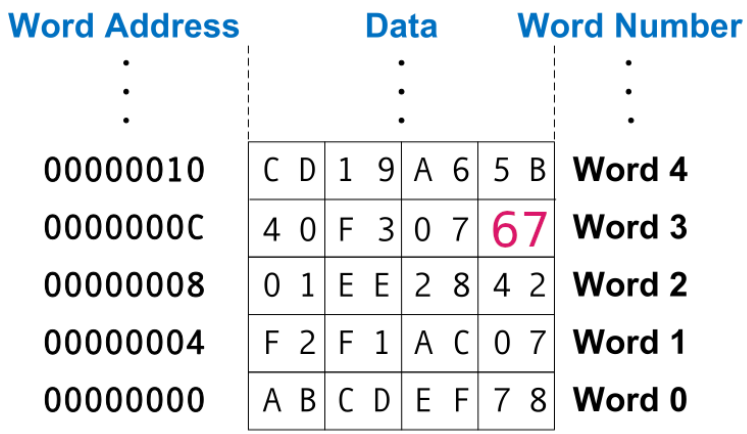
```
addi x6, x0, 0
lui x5, 0x01234
ori x5, x5, 0x567
sh x5, 12(x6)        # x5 → Mem[x6+12]
```



Save Byte

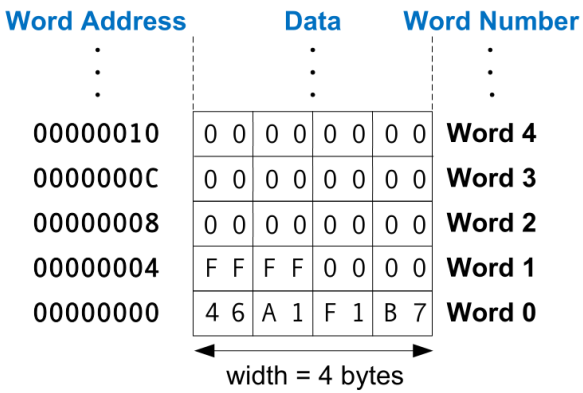
- Save byte example

```
addi x6, x0, 0
lui x5, 0x01234
ori x5, x5, 0x567
sb x5, 12(x6)      # x5 → Mem[x6+12]
```



Example

- Memory



Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

```
and s3, s1, s2
or  s4, s1, s2
xor s5, s1, s2
```

Result

s3	0100 0110	1010 0001	0000 0000	0000 0000
s4	1111 1111	1111 1111	1111 0001	1011 0111
s5	1011 1001	0101 1110	1111 0001	1011 0111

- Memory access example

High-Level Code

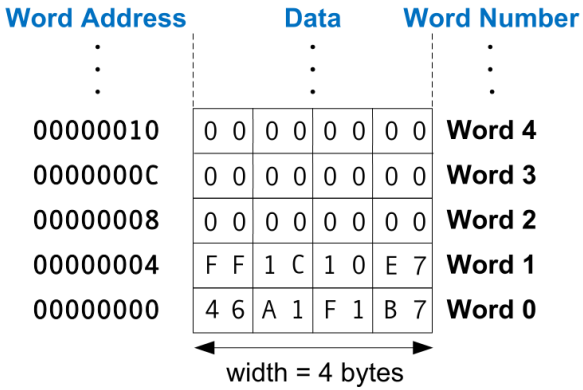
```
mem[2] = mem[1] & mem[0];
mem[3] = mem[1] | mem[0];
mem[4] = mem[1] ^ mem[0];
```

Assembly Code

```
lw   s1, 0(zero)    # s1 = 0x46A1F1B7
lw   s2, 4(zero)    # s2 = 0xFFFF0000
and  s3, s1, s2     # s3 = 0x46A10000
or   s4, s1, s2     # s4 = 0xFFFFF1B7
xor  s5, s1, s2     # s5 = 0xB95EF1B7
sw   s3, 8(zero)
sw   s4, 12(zero)
sw   s5, 16(zero)
```

Shift Example

- Memory



Assembly Code

```
slli t0, s5, 7
srli s1, s5, 17
srai t2, s5, 3
```

Source Register

s5	1111 1111	0001 1100	0001 0000	1110 0111

Result

t0	1000 1110	0000 1000	0111 0011	1000 0000
s1	0000 0000	0000 0000	0111 1111	1000 1110
t2	1111 1111	1110 0011	1000 0010	0001 1100

- Memory access example

High-Level Code

```
int mem[5];
mem[2] = mem[1] <<7;
mem[3] = (unsigned int)mem[1]>>17;
mem[4] = mem[1] >>3;
```

Assembly Code

```
lw   s5, 4(zero)     # s5 = 0xFF1C10E7
slli t0, s5, 7       # t0 = 0x8E087380
srli s1, s5, 17      # s1 = 0x00007F8E
srai t2, s5, 3       # t2 = 0xFFE3821C
sw   t0, 8(zero)
sw   s1, 12(zero)
sw   t2, 16(zero)
```

Jump Instruction

- `j` is the unconditional jump instruction

Assembly Code

```
    addi s0, zero, 4      # s0 = 4
    addi s1, zero, 1      # s1 = 1
    j sum                 # jump to sum
    addi s0, zero, 10     # s0 = 10
sum:
    add  s3, s0, s1       # s3 = s0 +s1
```

Conditional Branch

- beq, branch if equal

Assembly Code

```
    addi s0, zero, 4      # s0 = 4
    addi s1, zero, 1      # s1 = 1
    beq  s0, s1, sum      # if s0==s1, jump to sum
    addi s0, zero, 10     # s0 = 10
sum:
    add  s0, s0, s1       # s0 = s0 +s1
```

- After executing the above code, s0 = ?

Conditional Branch

- `bneq`, branch if not equal

Assembly Code

```
    addi  s0, zero, 4      # s0 = 4
    addi  s1, zero, 1      # s1 = 1
    bneq  s0, s1, sum      # if s0 !=s1, jump to sum
    addi  s0, zero, 10     # s0 = 10
sum:
    add   s0, s0, s1       # s0 = s0 +s1
```

- After executing the above code, `s0 = ?`

Conditional Branch

- `bltu`, branch if less than (unsigned comparison)

Assembly Code

```
    addi s0, zero,0xFFF  # s0 = 0xFFF
    addi s1, zero,0x123  # s1 = 0x123
    bltu s0, s1, sum     # if s0< s1, jump to sum
    addi s0, zero, 10    # s0 = 10
sum:
    add  s0, s0, s1      # s0 = s0 +s1
```

- After executing the above code, `s0 = ?`

Conditional Branch

- `blt`, branch if less than (signed comparison)

Assembly Code

```
    addi s0, zero,0xFFF  # s0 = 0xFFF
    addi s1, zero,0x123  # s1 = 0x123
    blt   s0, s1, sum    # if s0< s1, jump to sum
    addi s0, zero, 10    # s0 = 10
sum:
    add  s0, s0, s1      # s0 = s0 +s1
```

- After executing the above code, `s0 = ?`

Conditional Branch

- bgeu, branch if greater than or equal (unsigned comparison)

Assembly Code

```
    addi   s0, zero,0xFFF  # s0 = 0xFFF
    addi   s1, zero,0x123  # s1 = 0x123
    bgeu   s0, s1, sum     # if s0 >= s1, jump to sum
    addi   s0, zero, 10    # s0 = 10
sum:
    add    s0, s0, s1      # s0 = s0 + s1
```

- After executing the above code, s0 = ?

Conditional Branch

- bge, branch if greater than or equal (signed comparison)

Assembly Code

```
    addi   s0, zero,0xFFF  # s0 = 0xFFF
    addi   s1, zero,0x123  # s1 = 0x123
    bge    s0, s1, sum     # if s0 >= s1, jump to sum
    addi   s0, zero, 10    # s0 = 10
sum:
    add    s0, s0, s1      # s0 = s0 +s1
```

- After executing the above code, s0 = ?

Control Flow: If statement

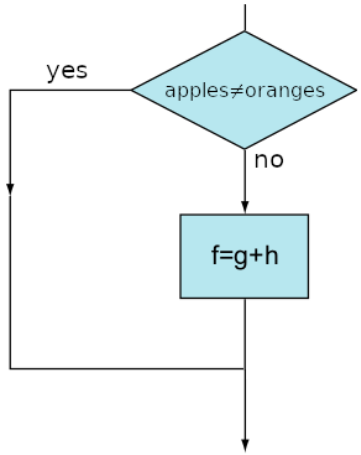
- if statement

High-Level Code

```
if (apples == oranges)
  f = g + h;
apples = oranges - h;
```

Assembly Code

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # skip if (apples != oranges)
    add s2, s3, s4 # f = g + h
L1: sub s0, s1, s4 # apples = oranges - h
```



Control Flow: If statement

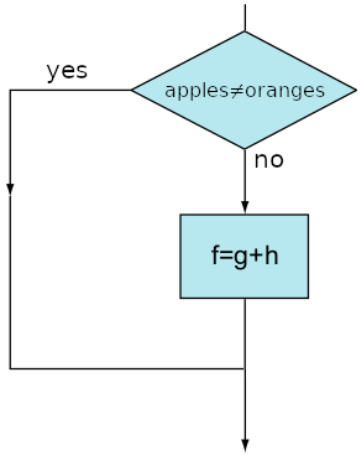
- if statement

High-Level Code

```
if (apples == oranges)
  f = g + h;
apples = oranges - h;
```

Assembly Code

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # skip if (apples != oranges)
    add s2, s3, s4 # f = g + h
L1: sub s0, s1, s4 # apples = oranges - h
```



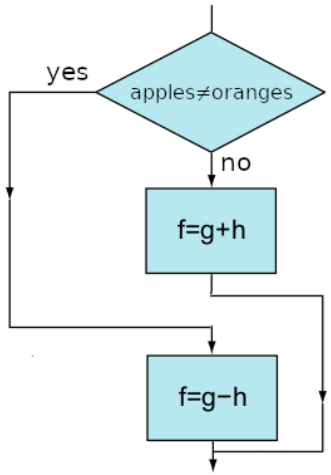
- if/else statement

High-Level Code

```
if (apples == oranges)
  f = g + h;
else
  f = g - h;
```

Assembly Code

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # skip if (apples != oranges)
    add s2, s3, s4 # f = g + h
    j   L2
L1: sub s2, s3, s4 # f = g - h
L2:
```



Control Flow: Switch statement

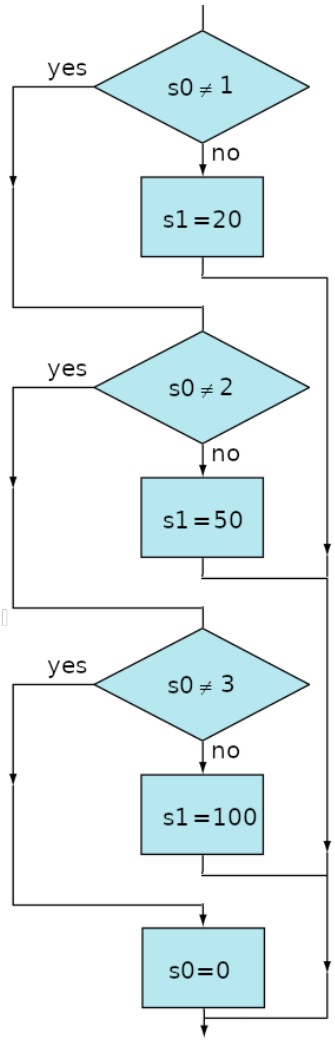
- switch as cascaded if statements

High-Level Code

```
switch (button) {
  case 1:  amt = 20; break;

  case 2:  amt = 50; break;

  case 3:  amt = 100; break;

  default: amt = 0;
}

// equivalent function using
// if/else statements
if      (button == 1)  amt = 20;
else if (button == 2)  amt = 50;
else if (button == 3)  amt = 100;
else                   amt = 0;
```

Assembly Code

```
# s0 = button, s1 = amt

case1:
 addi   t0, zero, 1       # t0 = 1
 bne    s0, t0, case2     # button == 1?
 addi   s1, zero, 20      # if yes, amt = 20
 j      done              # break out of case
case2:
 addi   t0, zero, 2       # t0 = 2
 bne    s0, t0, case3     # button == 2?
 addi   s1, zero, 50      # if yes, amt = 50
 j      done              # break out of case
case3:
 addi   t0, zero, 3       # t0 = 3
 bne    s0, t0, default   # button == 3?
 addi   s1, zero, 100     # if yes, amt = 100
 j      done              # break out of case
default:
 add    s1, zero, zero    # amt=0
done:
```



Loops: while

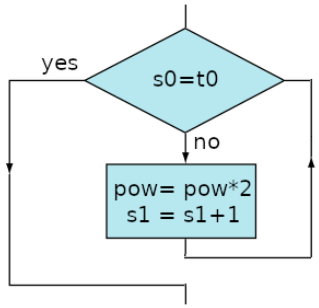
- while loop structure

High-Level Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


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

Assembly Code

```
#  s0 = pow, s1 = x
       addi s0,  zero, 1      # pow = 1
       add  s1,  zero, zero   # x = 0

       addi t0,  zero, 128    # t0 = 128
while: beq  s0,  t0, done     # pow = 128?
       slli s0,  s0, 1        # pow = pow * 2
       addi s1,  s1, 1        # x = x + 1
       j    while             # repeat loop
done:
```



Loops: while

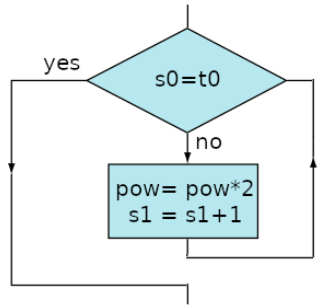
- while loop structure

High-Level Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


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

Assembly Code

```
#  s0 = pow, s1 = x
       addi s0,  zero, 1      # pow = 1
       add  s1,  zero, zero   # x = 0

       addi t0,  zero, 128    # t0 = 128
while: beq  s0,  t0, done     # pow = 128?
       slli s0,  s0, 1        # pow = pow * 2
       addi s1,  s1, 1        # x = x + 1
       j    while             # repeat loop
done:
```



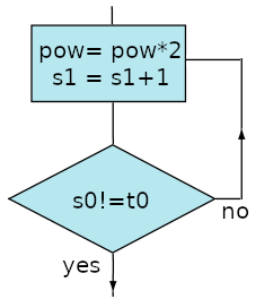
- do/while loop structure

High-Level Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

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

Assembly Code

```
#  s0 = pow, s1 = x
       addi s0,  zero, 1      # pow = 1
       add  s1,  zero, zero   # x = 0

       addi t0,  zero, 128    # t0 = 128
while: slli s0,  s0, 1        # pow = pow * 2
       addi s1,  s1, 1        # x = x + 1
       bne  s0,  t0, while    # pow = 128?
done:
```



For loop

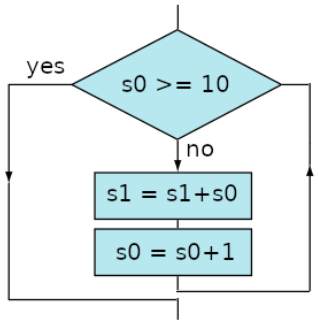
- For loop example

High-Level Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

Assembly Code

```
# s0 = i, s1 = sum
      addi  s1, zero, 0      # sum = 0
      addi  s0, zero, 0      # i = 0
      addi  t0, zero, 10     # t0 = 10
for:  bge   s0, t0, done     # i >= 10?
      add   s1, s1, s0       # sum = sum + i
      addi  s0, s0, 1        # i = i + 1
      j     for              # repeat loop
done:
```



Array Example

- Array Processing

High-Level Code

```
int i;
int scores[200];

for (i = 0; i < 200; i = i + 1)

  scores[i] = scores[i] + 10;
```

Assembly Code

```
# s0 = scores base address, s1 = i

  addi s1, zero, 0   # i = 0
  addi t2, zero, 200 # t2 = 200

for:
  bge  s1, t2, done  # if i >= 200 then done
  slli t0, s1, 2     # t0 = i * 4
  add  t0, t0, s0    # address of scores[i]
  lw   t1, 0(t0)     # t1 = scores[i]
  addi t1, t1, 10    # t1 = scores[i] + 10
  sw   t1, 0(t0)     # scores[i] = t1
  addi s1, s1, 1     # i = i + 1
  j    for           # repeat
done:
```

Address	Data
174303BC	scores[199]
174303B8	scores[198]
⋮	⋮
174300A4	scores[1]
174300A0	scores[0]

Assembler Directives (Reference)

- Assembler directives help guide the assembler
  - Assist in allocating and initializing global variables
  - Used to define constants
  - Assist in differentiating different memory segments

Assembler Directive	Description
`.text`	Text section
`.data`	Global data section
`.bss`	Global data initialized to 0
`.section .foo`	Section named `.foo`
`.align N`	Align next data/instruction on 2^N-byte boundary
`.balign N`	Align next data/instruction on N-byte boundary
`.globl sym`	Label `sym` is global
`.string "str"`	Store string "`str`" in memory
`.word w1,w2,...,wN`	Store N 32-bit values in successive memory words
`.byte b1,b2,...,bN`	Store N 8-bit values in successive memory bytes
`.space N`	Reserve N bytes to store variable
`.equ name,constant`	Define symbol `name` with value `constant`
`.end`	End of assembly code

ASCII Codes

- English language keyboard has fewer than 256 characters
- Each character can be coded in a byte
- American Standard Code for Information Interchange (ASCII) standardized this coding

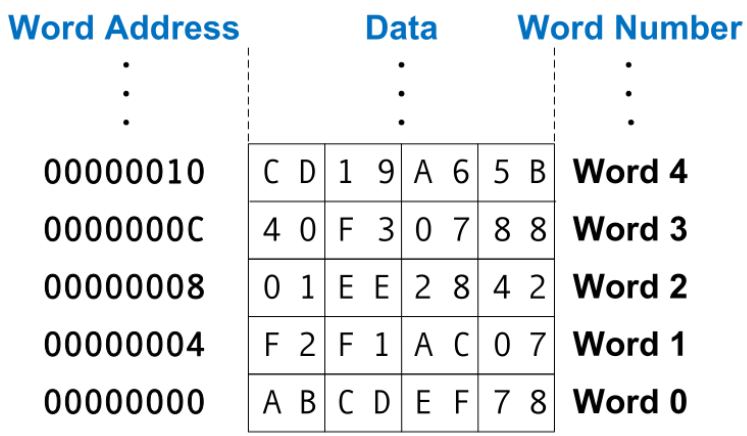
#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Load Byte (Unsigned)

- `lbu`, load the zero-extended byte

```
addi x6, x0, 0      # x6 ← 0
lbu x5, 12(x6)      # x5 ← Mem[x6+12]
```

x5 ← 0X00000088



- If we used `lb` we had,

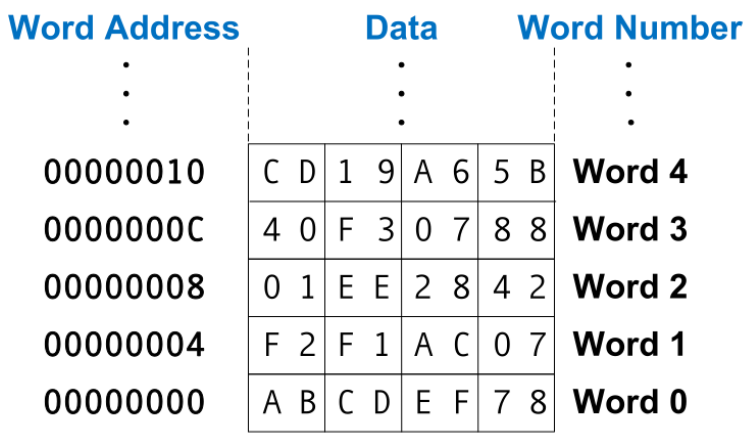
x5 ← 0XFFFFFF88

Load Half-Word (Unsigned)

- `lhu`, load the zero-extended half word

```
addi x6, x0, 0      # x6 ← 0
lhu x5, 4(x6)       # x5 ← Mem[x6+12]
```

x5 ← 0X0000AC07



- If we used `lh` we had,

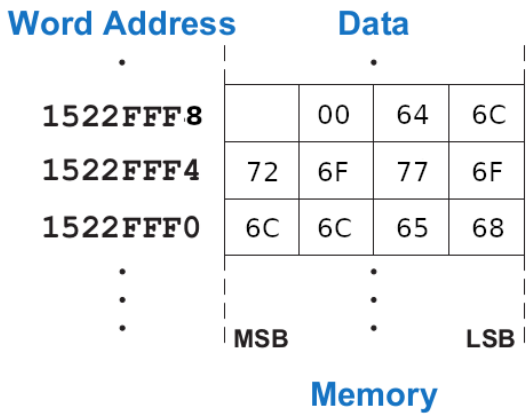
x5 ← 0XFFFFAC07

Strings

- Strings are null terminated byte arrays
- Simple example of string processing
- `la` load the base address of the array

Assembly Code

```
.data
 chararray:
    .string "helloworld"
.text
    la    s0, chararray    # s0 = base address of chararray
    addi s1, zero, 0       # i  =0
    addi t3, zero, 10      # t3 = 10
for:
    bge  s1, t3, done      # i>=10 ?
    add  t4, s0, s1        # t4= address of chararray[i]
    lbu   t5, 0(t4)        # t5= chararray[i]
    addi t5, t5, -32       # t5= chararray[i]-32
    sb    t5, 0(t4)        # chararray[i] = t5
    addi  s1, s1, 1        # i=i+1
    jal    zero, for       # repeat loop
done:
```



#	Char	#	Char	#	Char
20	space	40	@	60	`
21	!	41	A	61	a
22	"	42	B	62	b
23	#	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	'	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2A	*	4A	J	6A	j
2B	+	4B	K	6B	k
2C	,	4C	L	6C	l
2D	-	4D	M	6D	m
2E	.	4E	N	6E	n
2F	/	4F	O	6F	o
30	0	50	P	70	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x
39	9	59	Y	79	y
3A	:	5A	Z	7A	z
3B	;	5B	[	7B	{
3C	<	5C	\	7C	|
3D	=	5D	]	7D	}
3E	>	5E	^	7E	~
3F	?	5F	_		

System Calls (Optional)

- System calls are the services provided by the system mainly for input/output purposes
- To SYSCALL a system service
  1. Load the service number in register a7
  2. Load argument values, if any, in a0, a1, a2, a3, ... as specified.
  3. Issue the `ecall` instruction
  4. Retrieve return values, if any

Assembly Code

```
.data
 chararray:
    .string "helloworld"

.text
    la    s0, chararray    # s0 = base address of chararray
    addi s1, zero, 0       # i   =0
    addi t3, zero, 10      # t3 = 10
for:
    bge  s1, t3, done      # i>=10 ?
    add  t4, s0, s1        # t4= address of chararray[i]
    lbu   t5, 0(t4)        # t5= chararray[i]
    addi t5, t5, -32       # t5= chararray[i]-32
    sb    t5, 0(t4)        # chararray[i] = t5
    addi  s1, s1, 1        # i=i+1
    jal    zero, for       # repeat loop
done:
   addi a7, zero, 4        # Printing null-terminated string service
   add  a0, zero, s0       # Loading a0 with the string base address
   ecall
```

Instruction set (Reference)

- Arithmetic

Instruction	Example	Meaning	Comments
Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants

Instruction	Example	Meaning	Comments
multiply	mul x5, x6, x7	$x5 = (x6 * x7)_{31:0}$	multiply
multiply high	mulh x5, x6, x7	$x5 = (x6 * x7)_{63:32}$	multiply high signed signed
multiply high	mulhsu x5, x6, x7	$x5 = (x6 * x7)_{63:32}$	multiply high signed unsigned
multiply high	mulhu x5, x6, x7	$x5 = (x6 * x7)_{63:32}$	multiply high unsigned unsigned
divide signed	div x5, x6, x7	x5 = x6 / x7	divide signed signed
divide unsigned	divu x5, x6, x7	x5 = x6 / x7	divide unsigned unsigned
remainder signed	rem x5, x6, x7	x5 = x6%x7	remainder signed
remainder unsigned	remu x5, x6, x7	x5 = x6%x7	remainder unsigned

Instruction set (Reference)

- Bit Manipulation

Instruction	Example	Meaning	Comments
And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

Instruction set (Reference)

- Data Transfer

Instruction	Example	Meaning	Comments
Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits

Instruction set (Reference)

- Branches

Instruction	Example	Meaning	Comments
Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

Summary

- Operands: registers, memory and constants

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

- Registers set

Name	Register Number	Use
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	Temporary registers
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	Temporary registers

QUIZ 2

The second quiz covers material up to this slide

Registers

- **Zero**: zero constant
- **Saved registers** `s0-s11`: local variables (Saved)
- **Temporary registers** `t0-t6`: local variables (Temporary)
- **Argument registers** `a0-a7`: Function-call arguments
- **ra**: return address register
- **sp**: stack pointer
- **gp**: global pointer
- **tp**: thread pointer

Function Call

High-Level Code

```
int main() {
  simple();
  ...
}

// void means the function
// returns no value
void simple() {
  return;
}
```

Assembly Code

```
0x00000300 main:   jal  simple   # call function
0x00000304 ...
...        ...

0x0000051c simple: jr   ra       # return
```

- `jal <lablename>`: jump and link
  1. It saves the return address into `ra`
  2. Jumps to the first instruction after the given label
- `jr ra`
  - jumps back (return) to the saved address in `ra`

Passing arguments/Returning result

- By Convention
  - **Arguments**: a0 . . . a7
  - **Return value**: a0, (if needed) a1

High-Level Code

```
int main(){
  int y;
  . . .

  y = diffofsums(2, 3, 4, 5);
  . . .
}

int diffofsums(int f, int g, int h, int i){
  int result;

  result = (f + g) - (h + i);

  return result;
}
```

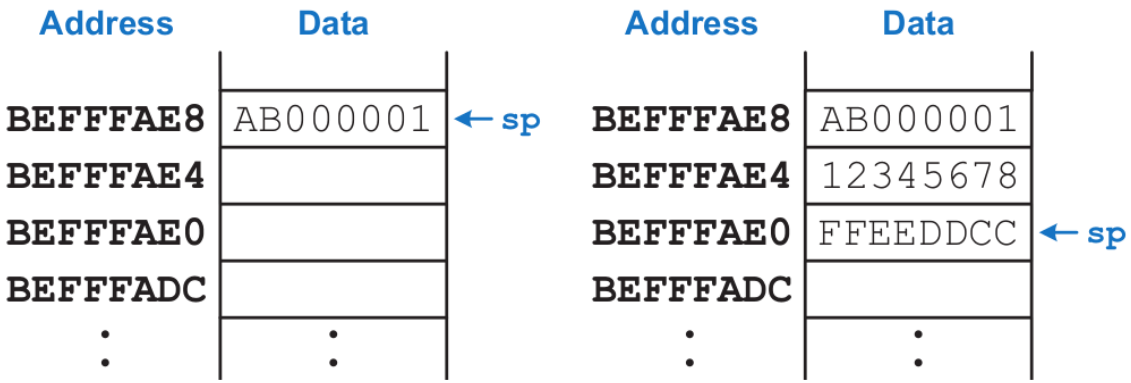
Assembly Code

```
# 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
```

- The function `diffofsums` has unintended side effects on s3!

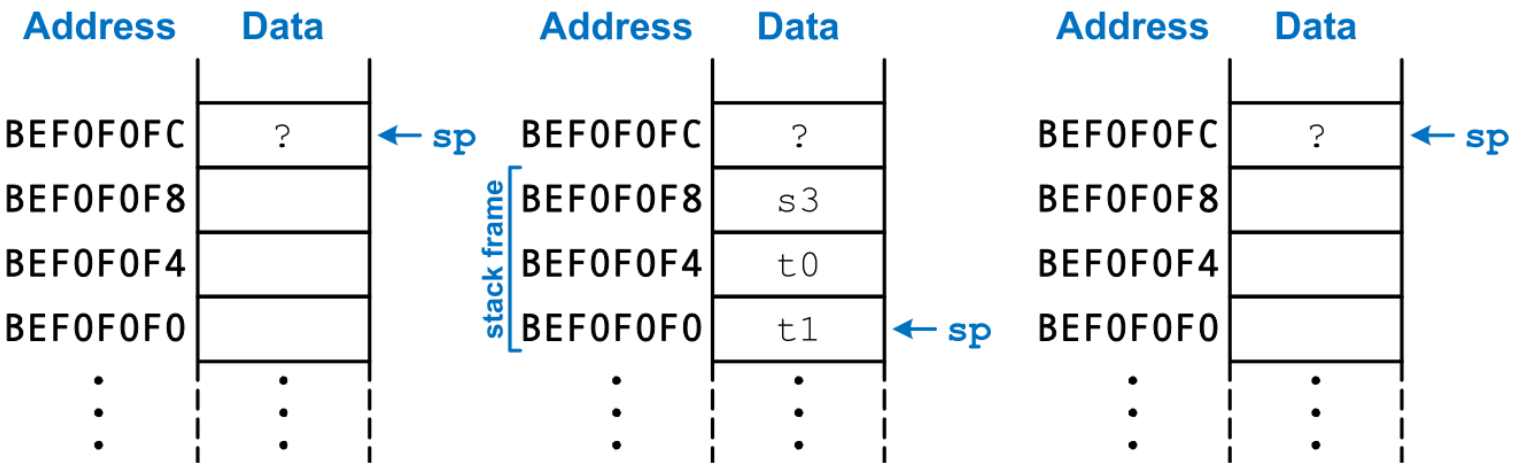
Stack and Stack pointer

- The stack is a portion of memory used as scratchpad
- By convention, the stack pointer sp always points to the current end of stack frame
- The stack pointer sp starts at a high memory address and decrements as needed
- Stack is used to save and restore registers that are used by a function




```
# sp=0XBEFFFAE8,   s0=0X12345678,   s1=0XFFEEDDCC
addi sp, sp, -8   # expanding the stack frame by 2 words
sw s0, 4(sp)      # pushing s0 into stack
sw s1, 0(sp)      # pushing s1 into stack
```

Stack as a Scratchpad



High-Level Code

```
int diffofsums(int f, int g, int h, int i)
 {
   int result;

   result = (f + g) - (h + i);

   return result;
 }
```

Assembly Code

```
# 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   s3, 8(sp)      # restore s3 from stack
   lw   t0, 4(sp)      # restore t0 from stack
   lw   t1, 0(sp)      # restore t1 from stack
   addi sp, sp, 12     # deallocate stack space
   jr   ra             # return to caller
```

Preserved/Temporary Registers

- **Caller Responsibility**: Saving necessary Nonpreserved registers
- **Callee Responsibility**: Saving Preserved registers

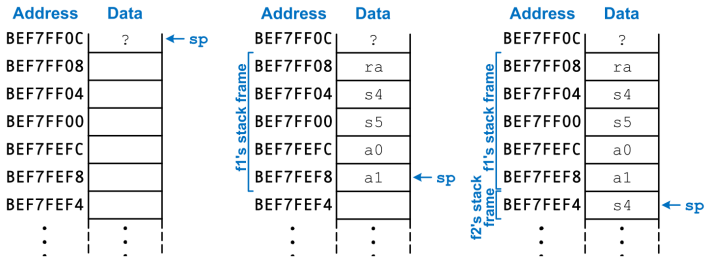
Preserved (*callee*-saved)	Nonpreserved (*caller*-saved)
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp`	
Stack above the stack pointer	Stack below the stack pointer

Assembly Code

```
# 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
```

Caller/Callee

- f1 is caller and callee
- f2 is a leaf function



High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);

  for (i = 0; i < a; i++)

     x = x + f2(b + i);

  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

Assembly Code

```
# a0 = a, a1 = b, s4 = i, s5 = x
f1:
   addi sp, sp, -12      # make room on stack for 3 registers
   sw   ra, 8(sp)        # save preserved registers used by f1
   sw   s4, 4(sp)
   sw   s5, 0(sp)
   add  s5, a0, a1       # x = (a + b)
   sub  t3, a0, a1       # temp = (a - b)
   mul  s5, s5, t3       # x = x * temp = (a + b) * (a - b)
   addi s4, zero, 0      # i = 0
for:
   bge  s4, a0, return   # if i >= a, exit loop
   addi sp, sp, -8       # make room on stack for 2 registers
   sw   a0, 4(sp)        # save nonpreserved regs. on stack
   sw   a1, 0(sp)
   add  a0, a1, s4       # argument is b + i
   jal  f2               # call f2(b + i)
   add  s5, s5, a0       # x = x + f2(b + i)
   lw   a0, 4(sp)        # restore nonpreserved registers
   lw   a1, 0(sp)
   addi sp, sp, 8
   addi s4, s4, 1        # i++
   j    for              # continue for loop
return:
   add  a0, zero, s5     # return value is x
   lw   ra, 8(sp)        # restore preserved registers
   lw   s4, 4(sp)
   lw   s5, 0(sp)
   addi sp, sp, 12       # restore sp
   jr   ra               # return from f1

# a0 = p,  s4 = r
f2:
   addi sp, sp, -4       # save preserved regs. used by f2
   sw   s4, 0(sp)
   addi s4, a0, 5        # r = p + 5
   add  a0, s4, a0       # return value is r + p
   lw   s4, 0(sp)        # restore preserved registers
   addi sp, sp, 4        # restore sp
   jr   ra               # return from f2
```

Programming Project (Optional)

- Write an assembly program to rearrange a given array of integers
- The rearrangement order is given by an array of indices
- Your code will start like this

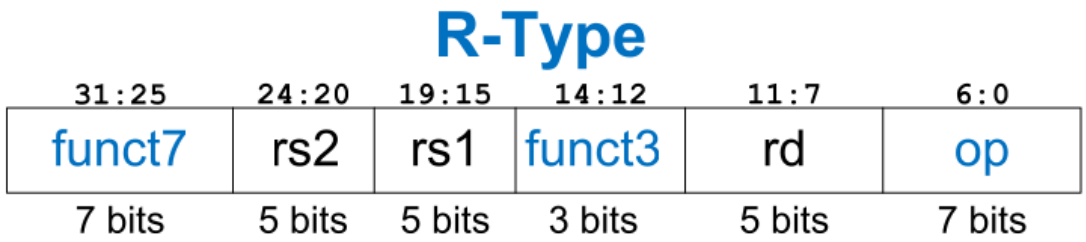
```
.data
   array:
    .word 10, 12, 9, 8, 1, 5, 23, 34, 0, 111
   indx:
    .word  1,  0, 3, 2, 5, 4,  9,  8, 7, 6

.text
   #### Your code goes here
```

- For example given the above numbers the array must be rearranged to, `12, 10, 8, 9, 5, 1, 111, 0, 34, 23`
- The project is optional and has extra bonus
- You must upload a working assembly file and a snapshot of the result
  - The assembly file: $rearrange.asm$
  - Result snapshot: $result.pdf$

Machine Language

- CPU only processes binary patterns
- Assembly programs must be translated into binary codes before execution
- In RISC-V architecture, each instruction is encoded as a 32-bit word
- RISC-V has the following main instruction formats
  - **R-Type:** three register operands
  - **I-Type**: one source, one destination and one 12-bit immediate operands
  - **Loads**: one source, one destination and one 12-bit immediate operands
  - **Stores**: two source registers and a 12-bits immediate operands
  - **Branches**: two source registers and one 12-bits (relative) immediate operands
  - **U/J Type**: one destination register and a 20-bits immediate operands

R-Type



- **R-Type bit-fields**
  - **op(6:0):** `0110011`
  - **funct3, funct7**: Identify the operation according to the table
  - **rd(11:7):** The destination register number
  - **rs1(19:15):** The `source1` register number
  - **rs2(24:20):** The `source2` register number

	funct7	funct3
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

R-Type (Examples)

	funct7	funct3
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

```
add s2, s3, s4
add x18,x19,x20
```

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51

funct7	rs2	rs1	funct3	rd	op
0000,000	10100	1001,1	000	10010	011,0011

(0x01498933)

R-Type (Examples)

	funct7	funct3
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

```
add s2, s3, s4
add x18,x19,x20
```

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51

funct7	rs2	rs1	funct3	rd	op
0000,000	10100	10011	000	10010	011,0011

(0x01498933)

```
sub t0, t1, t2
sub x5, x6, x7
```

funct7	rs2	rs1	funct3	rd	op
32	7	6	0	5	51

funct7	rs2	rs1	funct3	rd	op
0100,000	00111	00110	000	00101	011,0011

(0x407302B3)

R-Type (Examples)

	funct7	funct3
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

```
add s2, s3, s4
add x18,x19,x20
```

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51

funct7	rs2	rs1	funct3	rd	op
0000 000	10100	10011	000	10010	011 0011

(0x01498933)

```
sub t0, t1, t2
sub x5, x6, x7
```

funct7	rs2	rs1	funct3	rd	op
32	7	6	0	5	51

funct7	rs2	rs1	funct3	rd	op
0100 000	00111	00110	000	00101	011 0011

(0x407302B3)

```
sll s7, t0, s1
sll x23,x5, x9
```

funct7	rs2	rs1	funct3	rd	op
0	9	5	1	23	51

funct7	rs2	rs1	funct3	rd	op
0000 000	01001	00101	001	10111	011 0011

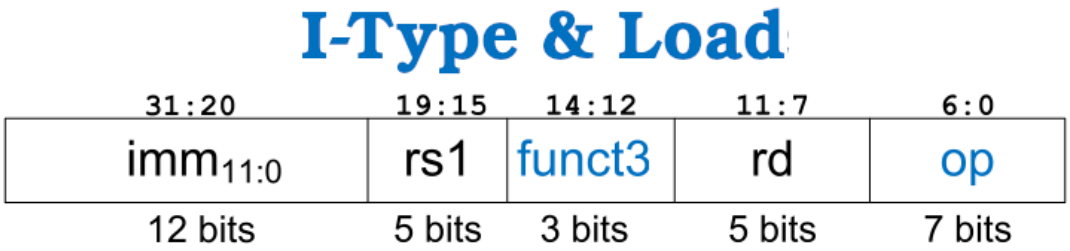
(0x00929BB3)

R-Type (Examples)

	funct7	funct3
add	0000000	000
sub	0100000	000
sll	0000000	001
slt	0000000	010
sltu	0000000	011
xor	0000000	100
srl	0000000	101
sra	0100000	101
or	0000000	110
and	0000000	111

	funct7	rs2	rs1	funct3	rd	op							
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
`sll s7, t0, s1` `sll x23,x5, x9`	0	9	5	1	23	51	0000 000	01001	00101	001	10111	011 0011	(0x00929BB3)
`xor s8, s9, s10` `xor x24,x25,x26`	0	26	25	4	24	51	0000 000	11010	11001	100	11000	011 0011	(0x01ACCC33)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

I-Type & Loads



- **I-Type bit-fields**
  - **op(6:0):** 0010011
  - **funct3**: Identify the operation according to the table
  - **rd(11:7):** The destination register number
  - **rs1(19:15):** The source1 register number
  - **imm(31:20):** The Immediate in Two's complement

	funct3
addi	000
slli	001
slti	010
sltiu	011
xori	100
srli	101 $bit_{30}=0$
srai	101 $bit_{30}=1$
ori	110
andi	111

- **Load bit-fields**
  - **op(6:0):** 0000011
  - **funct3**: Identify the operation according to the table
  - **rd(11:7):** The destination register number
  - **rs1(19:15):** The source1 register number
  - **imm(31:20):** The Immediate in Two's complement

	funct3
lb	000
lh	001
lw	010
lbu	100
lhu	101

I-Type (Examples)

	funct3	
addi	000	
slli	001	
slti	010	
sltiu	011	
xori	100	
srli	101	$bit_{30}=0$
srai	101	$bit_{30}=1$
ori	110	
andi	111	

```
slli s2, s7, 5
slli x18, x23, 5
```

$imm_{11:0}$	rs1	funct3	rd	op
5	23	1	18	19
0000 0000 0101	10111	001	10010	001 0011

(0x005B9913)

I-Type (Examples)

	funct3	
addi	000	
slli	001	
slti	010	
sltiu	011	
xori	100	
srli	101	$bit_{30} = 0$
srai	101	$bit_{30} = 1$
ori	110	
andi	111	

```
slli s2, s7, 5
slli x18, x23, 5
```

$imm_{11:0}$	rs1	funct3	rd	op
5	23	1	18	19
12 bits	5 bits	3 bits	5 bits	7 bits

$imm_{11:0}$	rs1	funct3	rd	op	
0000 0000 0101	10111	001	10010	001 0011	**(0x005B9913)**
12 bits	5 bits	3 bits	5 bits	7 bits	

```
srai t1, t2, 29
srai x6, x7, 29
```

29	7	5	6	19	0100 0001 1101	00111	101	00110	001 0011	**(0x41D3D313)**
12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Loads (Examples)

	funct3
lb	000
lh	001
lw	010
lbu	100
lhu	101

```
lw   t2, -6(s3)
lw   x7, -6(x19)
```

$imm_{11:0}$	rs1	funct3	rd	op
-6	19	2	7	3

$imm_{11:0}$	rs1	funct3	rd	op	
1111 1111 1010	10011	010	00111	000 0011	**(0xFFA9A383)**

Loads (Examples)

	funct3
lb	000
lh	001
lw	010
lbu	100
lhu	101

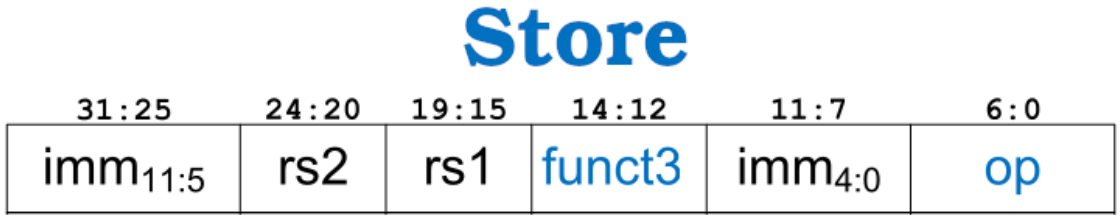
```
lw   t2, -6(s3)
lw   x7, -6(x19)
```

$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	
-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	**(0xFFA9A383)**

```
lb   s4, 0x1F(s4)
lb   x20,0x1F(x20)
```

$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	
0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	**(0x01FA0A03)**
12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Store Instructions



- **Store bit-fields**
  - **op(6:0):** `0100011`
  - **funct3**: Identify the operation according to the table
  - **rs1(19:15):** The `source1` register number
  - **rs2(24:20):** The `source2` register number
  - **imm(31:25)(11:7):** 12-bits immediate

	funct3
sb	000
sh	001
sw	010

Store (Examples)

	funct3
sb	000
sh	001
sw	010

```
sw t2, -6(s3)
sw x7, -6(x19)
```

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
1111 111	7	19	2	11010	35

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
1111 111	00111	10011	010	11010	010 0011	**(0xFE79AD23)**

Store (Examples)

	funct3
sb	000
sh	001
sw	010

```
sw t2, -6(s3)
sw x7, -6(x19)
```

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
1111 111	7	19	2	11010	35

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
1111 111	00111	10011	010	11010	010 0011

(0xFE79AD23)

```
sh s4, 23(t0)
sh x20,23(x5)
```

0000 000	20	5	1	10111	35

0000 000	10100	00101	001	10111	010 0011

(0x01429BA3)

Store (Examples)

	funct3
sb	000
sh	001
sw	010

```
sw t2, -6(s3)
sw x7, -6(x19)
```

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	**(0xFE79AD23)**

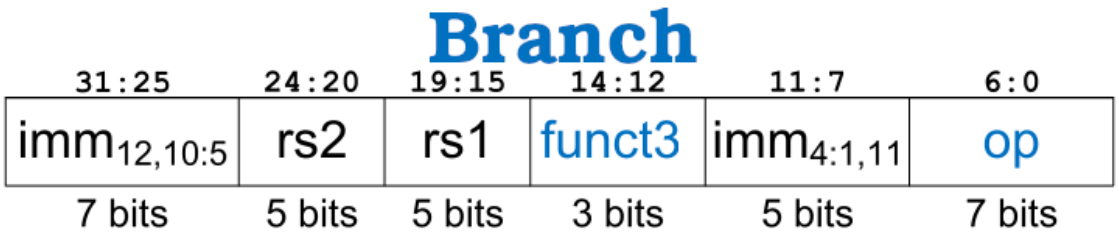
```
sh s4, 23(t0)
sh x20,23(x5)
```

0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	**(0x01429BA3)**

```
sb t5, 0x2D(zero)
sb x30,0x2D(x0)
```

0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	**(0x03E006A3)**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Branch Instructions



- **Store bit-fields**
  - **op(6:0):** 1100011
  - **funct3**: Identify the operation according to the table
  - **rs1(19:15):** The source1 register number
  - **rs2(24:20):** The source2 register number
  - **imm(31)(7)(30:25)(11:8):** 12-bit immediate

	funct3
beq	000
bne	001
blt	100
bge	101
bltu	110
bgeu	111

Branch (Examples)

```
0x70   L2: beq  s0, t5, L1
0x74       add  s1, s2, s3
0x78       sub  s5, s6, s7
0x7C       lw   t0, 0(s1)
0x80   L1: addi s1, s1, -15
0x84       bne  s1, t3, L2
```

	funct3
beq	000
bne	001
blt	100
bge	101
bltu	110
bgeu	111

$imm_{12:0}$ = 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

```
beq s0, t5,  L1
beq x8, x30, 16
```

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Branch (Examples)

```
0x70   L2: beq  s0, t5, L1
0x74       add  s1, s2, s3
0x78       sub  s5, s6, s7
0x7C       lw   t0, 0(s1)
0x80   L1: addi s1, s1, -15
0x84       bne  s1, t3, L2
```

	funct3
beq	000
bne	001
blt	100
bge	101
bltu	110
bgeu	111

$imm_{12:0}$ = 16 — 0 | 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

beq s0, t5, L1
beq x8, x30, 16

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x01E40863)

$imm_{12:0}$ = -20 — 1 | 1 | 1 1 1 1 | 1 1 1 0 | 1 1 0 0

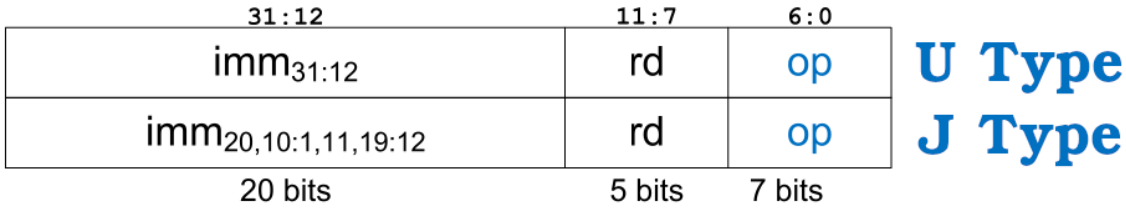
bit number — 12 | 11 | 10 9 8 7 | 6 5 4 3 | 2 1 0

bne s1, t3, L2
bne x9, x28, L2

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1111 111	28	9	1	0110 1	99	1111 111	11100	01001	001	0110 1	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x FFC496E3)

U/J-Type Instructions



- **U-Type bit-fields**
  - **op(6:0):**
    - `lui:` `0110111`
    - `auipc:` `0010111`
  - **rd(11:7):** The `destination` register number
  - **imm(31:12):** 20-bits immediate

- **J-Type bit-fields**
  - **op(6:0):**
    - `jal:` `1101111`
  - **rd(11:7):** The `destination` register number
  - **imm(31)(19:12)(20)(30:21):** 21-bits immediate!

U/J-Type (Examples)

```
0x0000540C          jal ra, func1
0x00005410          add s1, s2, s3
...                 ...

0x000ABC04  func1:  add s4, s5, s8
```

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	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

Assembly

```
jal ra, func1
jal x1, 0xA67F8
```

Field Values

$imm_{20,10:1,11,19:12}$	rd	op
0111 1111 1000 1010 0110	1	111
20 bits	5 bits	7 bits

Machine Code

$imm_{20,10:1,11,19:12}$	rd	op
0111 1111 1000 1010 0110	00001	110 1111
20 bits	5 bits	7 bits

(0x7F8A60EF)

U/J-Type (Examples)

```
0x0000540C          jal ra, func1
0x00005410          add s1, s2, s3
...                 ...

0x000ABC04  func1:  add s4, s5, s8
```

imm = 0xA67F8	0		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

Assembly | **Field Values** | **Machine Code**

`jal ra, func1`
`jal x1, 0xA67F8`

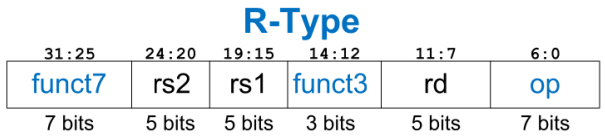
	$imm_{20,10:1,11,19:12}$	rd	op		$imm_{20,10:1,11,19:12}$	rd	op	
	0111 1111 1000 1010 0110	1	111		0111 1111 1000 1010 0110	00001	110 1111	(0x7F8A60EF)
	20 bits	5 bits	7 bits		20 bits	5 bits	7 bits	

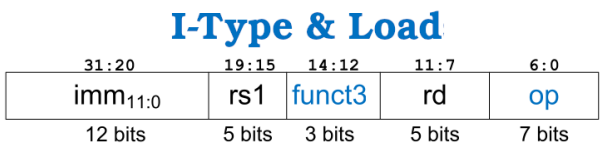
`lui s5, 0x8CDEF`
`lui x21,0x8CDEF`

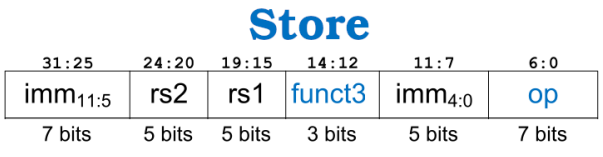
	$imm_{31:12}$	rd	op		$imm_{31:12}$	rd	op	
	0x8CDEF	21	55		1000 1100 1101 1110 1111	10101	011 0111	(0x8CDEFAB7)
	20 bits	5 bits	7 bits		20 bits	5 bits	7 bits	

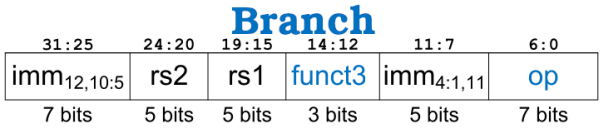
Summary

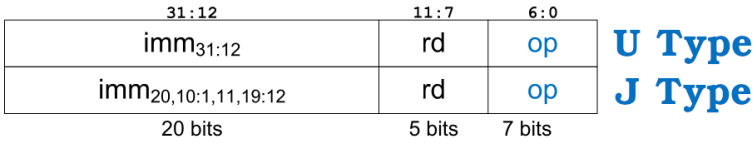
- Instruction types
  - **R-Type**
  - **I-Type**
  - **Loads**
  - **Stores**
  - **Branches**
  - **U/J Type**

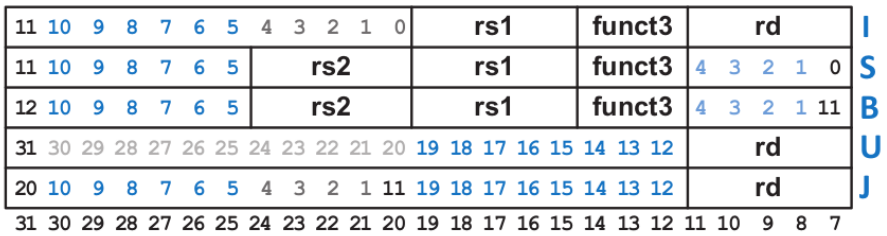










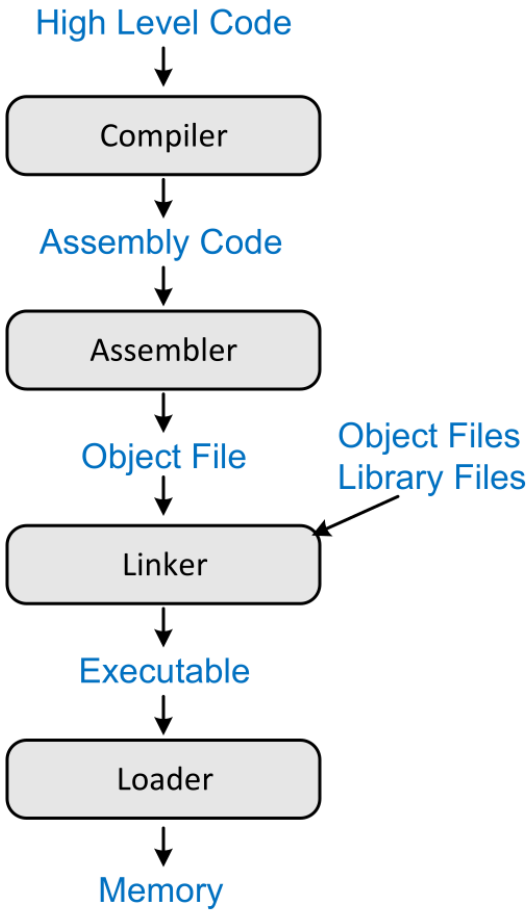


Instruction Machine Codes (Reference)

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 − rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

Compiling, Assembling and Loading

- **Compiler:** *High level language* $\rightarrow$ *assembly language*
- **Assembler:** *Assembly* $\rightarrow$ *machine code* (*obj file*)
- **Linker:** *Unifies obj files + figures out addresses*
- **Loader:** *Loads program into memory and excutes*



Memory Map

- RISC-V does not define any specific Memory Map
- **Operating System and I/O**: Refers to the memory-mapped regions dedicated to interacting with input/output devices
- **Dynamic Data**: Represents the heap section of memory used for dynamic memory allocation
- **Global Data**: Refers to the segment of memory where global and static variables are stored
- **Text**: The memory segment containing the executable code or instructions of a program
- **Exception Handlers**: Memory regions where routines for handling exceptions (e.g., interrupts or system errors) are stored