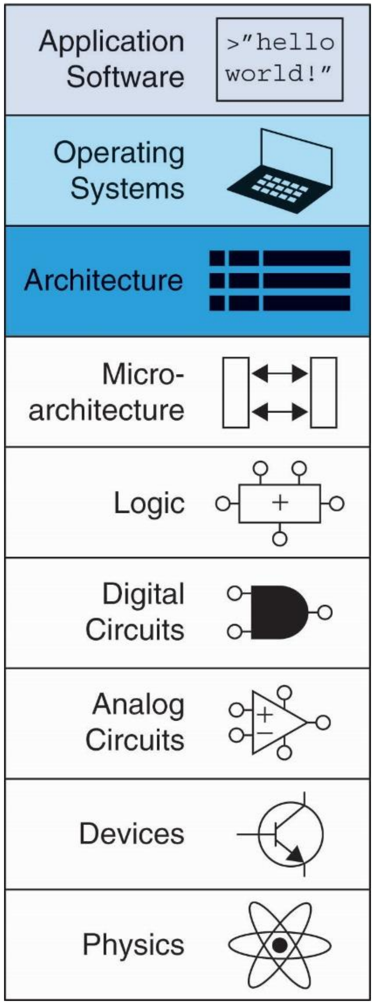
RISC-V Architecture
&
Processor Design

Week 3

Instruction Set Architectures
and
RISC-V

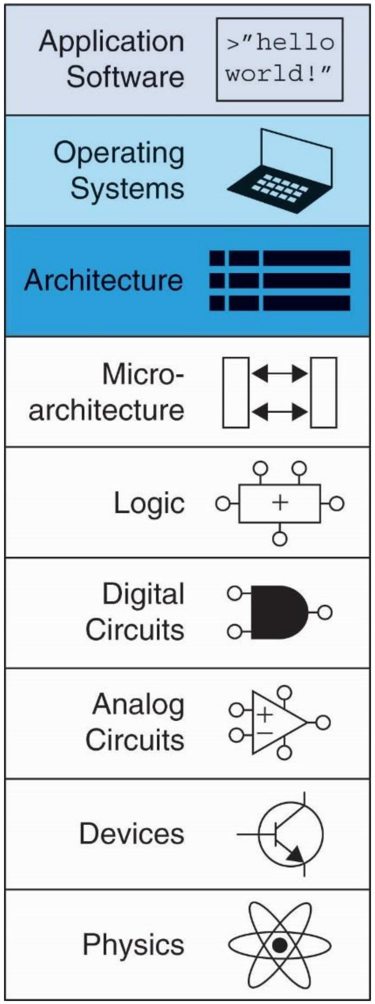
Topics

- **Introduction**
- **Assembly Language**
- **Programming**
- **Machine Language**
- **Addressing Modes**
- **Lights, Camera, Action: Compiling, Assembly, & Loading**
- **Odds & Ends**

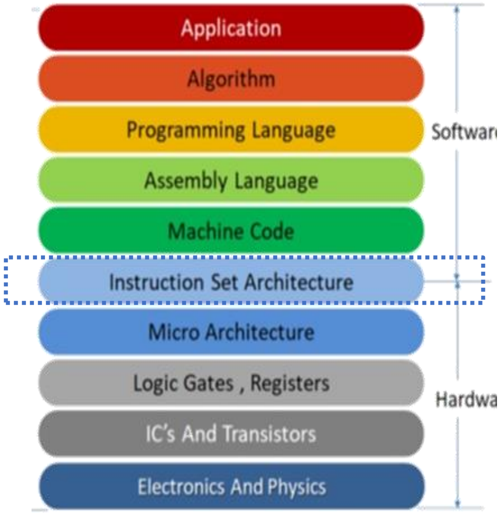


Introduction

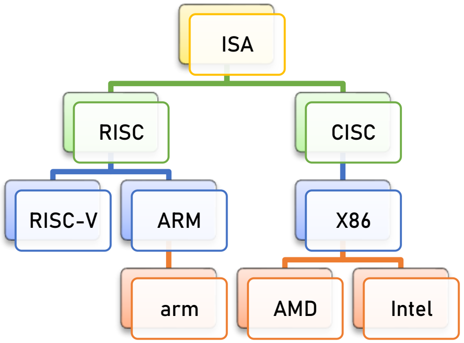
- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware



Instruction Set Architectures



ISA: Instruction Set Architecture
A standard interface between the hardware and software. ISAs can effect the structure and performance of the hardware.



Instruction Set Architecture (ISA)

RISC-V
arm

RISC

Reduced Instruction Set Computing

Simple instructions
Relies more on **compiler optimizations**
Many instructions for more complex functions
Simpler and smaller core

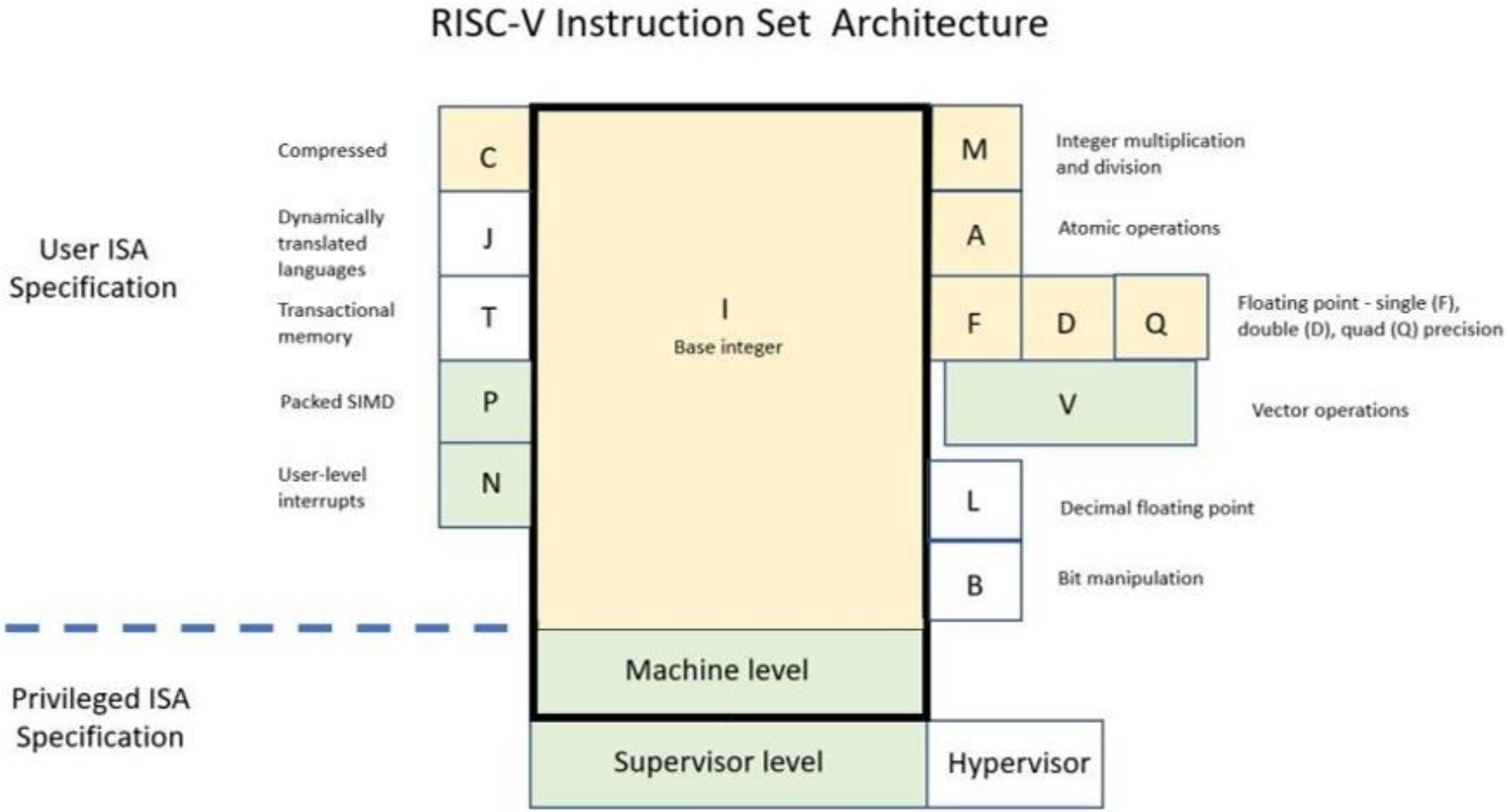
X86

CISC

Complex Instruction Set Computing

Complex instructions
Less pressure on the compiler
Single instruction can handle complex functions
Complex and bigger core

RISC-V ISA Overview



Complete specifications; https://riscv.org/technical/specifications/

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **RISC-V architecture:**
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
  - First widely accepted open-source computer architecture

Once you've learned one architecture, it's easier to learn others

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Instructions

Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```

- **`add:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

Instructions: Subtraction

Similar to addition - only **mnemonic** changes

C Code

```
a = b - c;
```

RISC-V assembly code

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

- **`sub:`** mnemonic
- **`b, c:`** source operands
- **`a:`** destination operand

Design Principle 1

Simplicity favors regularity

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

Multiple Instructions

More complex code is handled by multiple RISC-V instructions.

C Code

```
a = b + c - d;
```

RISC-V assembly code

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

Design Principle 2

Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Operands

Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- RISC-V includes only a small number of registers

RISC-V Register Set

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

Operands: Registers

- **Registers:**
  - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
  - Using name is preferred
- Registers used for **specific purposes**:
  - `zero` always holds the **constant value 0**.
  - the ***saved registers***, `s0-s11`, used to hold variables
  - the ***temporary registers***, `t0-t6`, used to hold intermediate values during a larger computation
  - Discuss others later

Instructions with Registers

- Revisit `add` instruction

C Code

```
a = b + c;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

indicates a single-line comment

Instructions with Constants

- `addi` instruction

C Code

```
a = b + 6;
```

RISC-V assembly code

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

Memory Operands

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

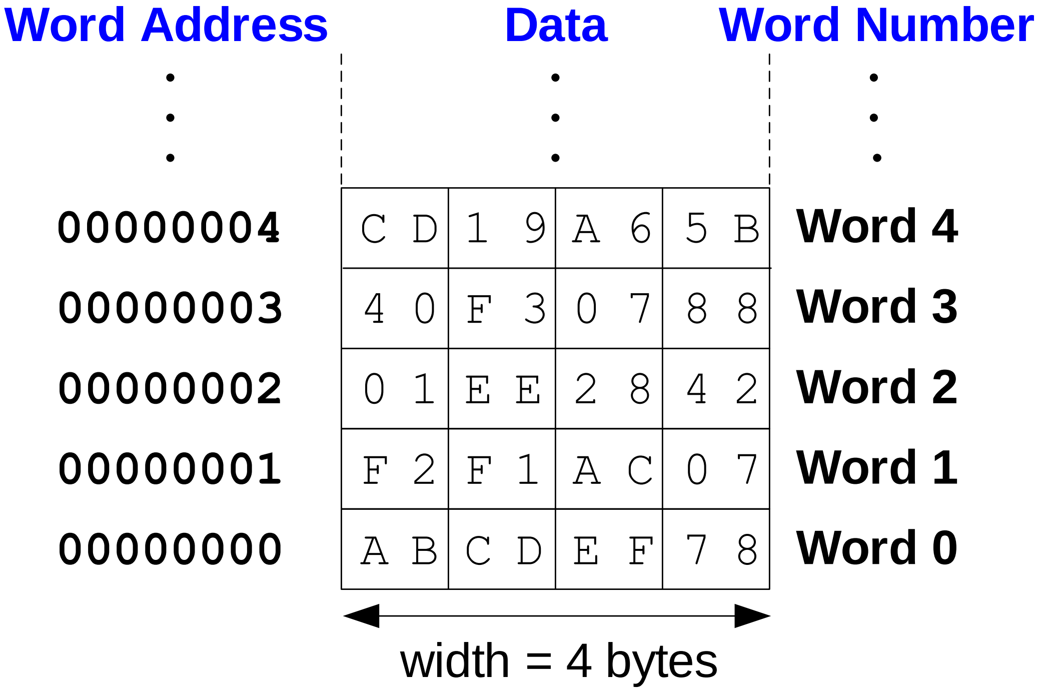
Memory

- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is **byte-addressable**

Word-Addressable Memory

- Each 32-bit data word has a unique address



Word Address	Data				Word Number
⋮		⋮			⋮
00000004	C D	1 9	A 6	5 B	Word 4
00000003	4 0	F 3	0 7	8 8	Word 3
00000002	0 1	E E	2 8	4 2	Word 2
00000001	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

RISC-V uses **byte-addressable** memory, which we'll talk about next.

Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

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

- **Address calculation:**
  - add *base address* (`s0`) to the *offset* (5)
  - address = (`s0` + 5)
- **Result:**
  - `t1` holds the data value at address (`s0` + 5)

Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `s3`
  - address = (0 + 1) = 1
  - `s3` = 0xF2F1AC07 after load

Assembly code

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

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	**F 2 F 1 A C 0 7**	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory write is called a ***store***
- **Mnemonic:** *store word* (`sw`)

Writing Word-Addressable

- **Example:** Write (store) the value in `t4` into memory address 3
  - add the base address (`zero`) to the offset (0x3)
  - address: (0 + 0x3) = 3
  - for example, if `t4` holds the value 0xFEEDCABB, then after this instruction completes, word 3 in memory will contain that value

Offset can be written in **decimal** (default) or **hexadecimal**

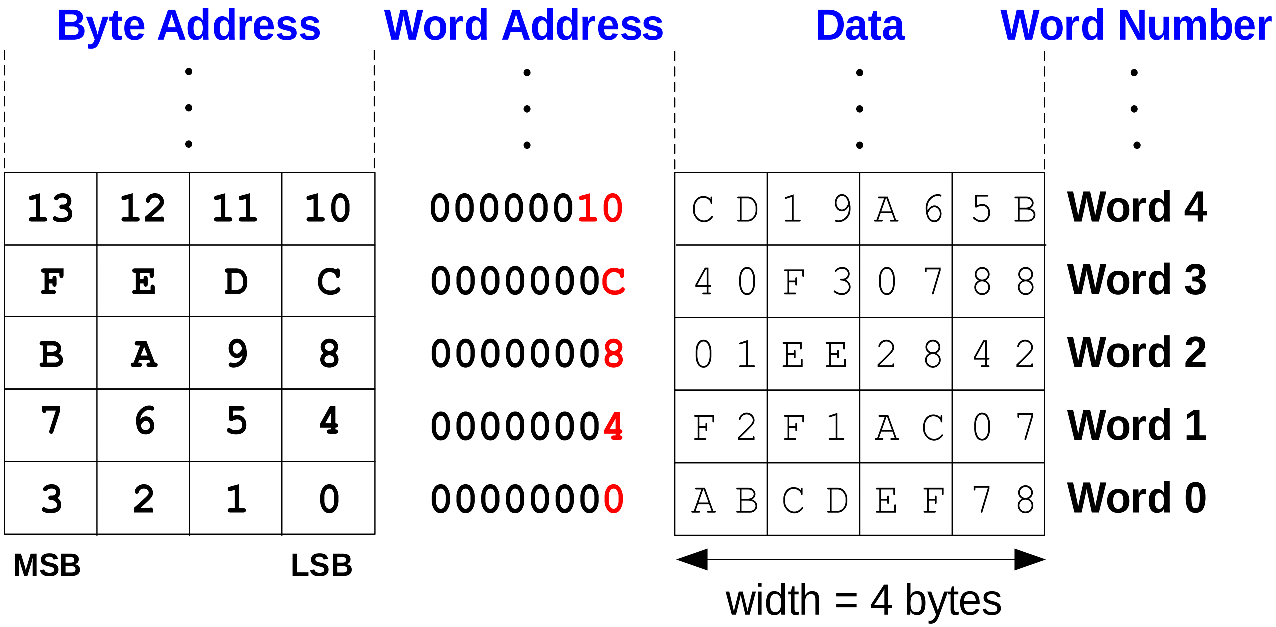
Assembly code

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

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	CD19A65B	Word 4
00000003	**FEEDCABB**	Word 3
00000002	01EE2842	Word 2
00000001	F2F1AC07	Word 1
00000000	ABCDEF78	Word 0

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address **increments by 4**



Byte Address				Word Address	Data				Word Number
⋮				⋮	⋮				⋮
13	12	11	10	00000010	CD	19	A6	5B	Word 4
F	E	D	C	0000000C	40	F3	07	88	Word 3
B	A	9	8	00000008	01	EE	28	42	Word 2
7	6	5	4	00000004	F2	F1	AC	07	Word 1
3	2	1	0	00000000	AB	CD	EF	78	Word 0
MSB			LSB						

width = 4 bytes

Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)

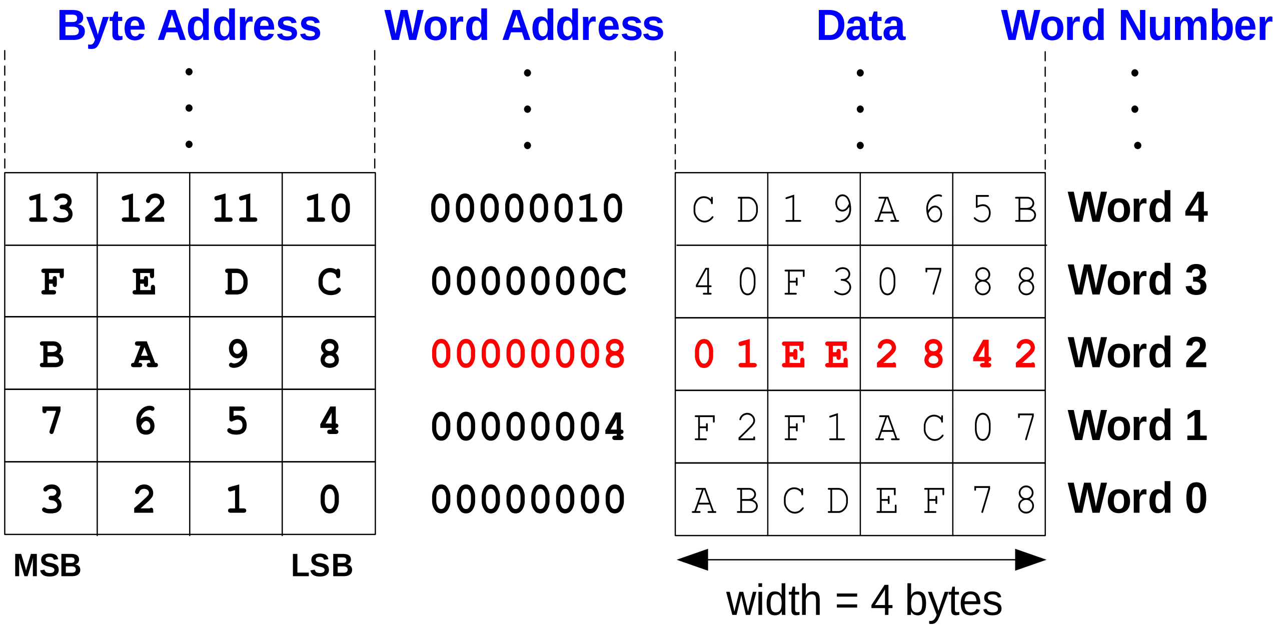
- RISC-V is **byte-addressed**, not word-addressed

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into `s3`.
- `s3` holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```



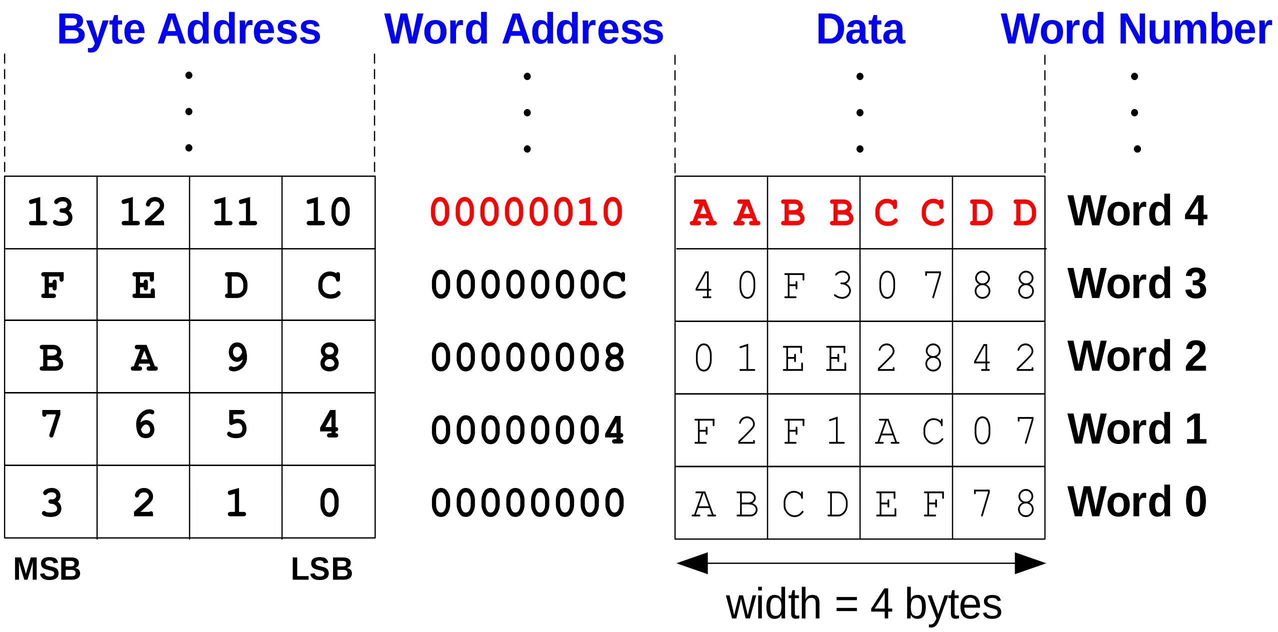
Byte Address				Word Address	Data				Word Number
⋮				⋮	⋮				⋮
13	12	11	10	00000010	C D	1 9	A 6	5 B	Word 4
F	E	D	C	0000000C	4 0	F 3	0 7	8 8	Word 3
B	A	9	8	00000008	0 1	E E	2 8	4 2	Word 2
7	6	5	4	00000004	F 2	F 1	A C	0 7	Word 1
3	2	1	0	00000000	A B	C D	E F	7 8	Word 0
MSB			LSB						

Writing Byte-Addressable Memory

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

RISC-V assembly code

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



Byte Address				Word Address	Data				Word Number
⋮				⋮	⋮				⋮
13	12	11	10	00000010	AA	BB	CC	DD	Word 4
F	E	D	C	0000000C	40	F3	07	88	Word 3
B	A	9	8	00000008	01	EE	28	42	Word 2
7	6	5	4	00000004	F2	F1	AC	07	Word 1
3	2	1	0	00000000	AB	CD	EF	78	Word 0

Generating Constants

Generating 12-Bit Constants

- 12-bit signed constants (immediates) using `addi`:

C Code

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

RISC-V assembly code

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

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 and 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`

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
```

Logical / Shift Instructions

Programming

- **High-level languages:**
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- **High-level constructs:** loops, conditional statements, arrays, function calls
- **First, introduce instructions that support these:**
  - Logical operations
  - Shift instructions
  - Multiplication & division
  - Branches & Jumps

Logical Instructions

- **`and`, `or`, `xor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:

      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:

      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `xor`: useful for **inverting** bits:
    - A XOR -1 = NOT A   (remember that -1 = 0xFFFFFFFF)

Logical Instructions: Example 1

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

Assembly Code		Result			
`and s3, s1, s2`	s3	0100 0110	1010 0001	0000 0000	0000 0000
`or  s4, s1, s2`	s4	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions: Example 2

Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100

← sign-extended →

Assembly Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

Result

s5								
s6								
s7								

-1484 = **0xA34** in 12-bit 2's complement representation.

Shift Instructions

Shift amount is in (lower 5 bits of) a register

- `sll:` shift left logical
  - **Example:** `sll t0, t1, t2 # t0 = t1 << t2`
- `srl:` shift right logical
  - **Example:** `srl t0, t1, t2 # t0 = t1 >> t2`
- `sra:` shift right arithmetic
  - **Example:** `sra t0, t1, t2 # t0 = t1 >>> t2`

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- `slli:` shift left logical immediate
  - **Example:** `slli t0, t1, 23 # t0 = t1 << 23`
- `srli:` shift right logical immediate
  - **Example:** `srli t0, t1, 18 # t0 = t1 >> 18`
- `srai:` shift right arithmetic immediate
  - **Example:** `srai t0, t1, 5 # t0 = t1 >>> 5`

Multiplication and Division

Multiplication

32 × 32 multiplication → 64 bit result

```
mul s3, s1, s2
```

s3 = lower 32 bits of result

```
mulh s4, s1, s2
```

s4 = upper 32 bits of result, treats operands as signed

{s4,s3} = s1 x s2

Example: s1 = 0x40000000 = 2^{30}; s2 = 0x80000000 = -2^{31}

s1 x s2 = -2^{61} = 0xE0000000 00000000

s4 = 0xE0000000; s3 = 0x00000000

Division

32-bit division → 32-bit quotient & remainder

```
– div  s3, s1, s2  # s3 = s1/s2
– rem  s4, s1, s2  # s4 = s1%s2
```

Example: s1 = 0x00000011 = 17; s2 = 0x00000003 = 3

s1 / s2 = 5

s1 % s2 = 2

s3 = 0x00000005; s4 = 0x00000002

Branches & Jumps

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
    - jump and link register (`jalr`)

We'll talk about these when discuss function calls

Conditional Branching

```
# RISC-V assembly
 addi s0, zero, 4        # s0 = 0 + 4 = 4
 addi s1, zero, 1        # s1 = 0 + 1 = 1
 slli s1, s1, 2          # s1 = 1 << 2 = 4
 beq  s0, s1, target     # branch is taken
 addi s1, s1, 1          # not executed
 sub  s1, s1, s0         # not executed

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

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

The Branch Not Taken (bne)

```
# RISC-V assembly
    addi    s0, zero, 4         # s0 = 0 + 4 = 4
    addi    s1, zero, 1         # s1 = 0 + 1 = 1
    slli    s1, s1, 2           # s1 = 1 << 2 = 4
    bne     s0, s1, target      # branch not taken
    addi    s1, s1, 1           # s1 = 4 + 1 = 5
    sub     s1, s1, s0          # s1 = 5 - 4 = 1

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

Unconditional Branching (j)

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

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

Conditional Statements & Loops

Conditional Statements & Loops

- **Conditional Statements**
  - `if` statements
  - `if/else` statements
- **Loops**
  - `while` loops
  - `for` loops

If Statement

C Code

```
if (i == j)
  f = g + h;


f = f - i;
```

RISC-V assembly code

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

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If/Else Statement

C Code

```
if (i == j)
  f = g + h;


else
  f = f - i;
```

RISC-V assembly code

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

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

While Loops

C 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;
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x
```

Assembly tests opposite case (`pow == 128`) of high-level code (`pow != 128`)

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes **before** the loop begins
- `condition:` is tested **at the beginning** of each iteration
- `loop operation:` executes at the end **end** of each iteration
- `statement:` executes **each time** the condition is met

For Loops

C Code

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

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

RISC-V assembly code

```
# s0 = i, s1 = sum
```

Less Than Comparison

C Code

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

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

RISC-V assembly code

```
# s0 = i, s1 = sum
```

Less Than Comparison: Version 2

C Code

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

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

RISC-V assembly code

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

slt: set if less than instruction

```
slt t2, s0, t0   # if s0 < t0, t2 = 1
                 # otherwise t2 = 0
```

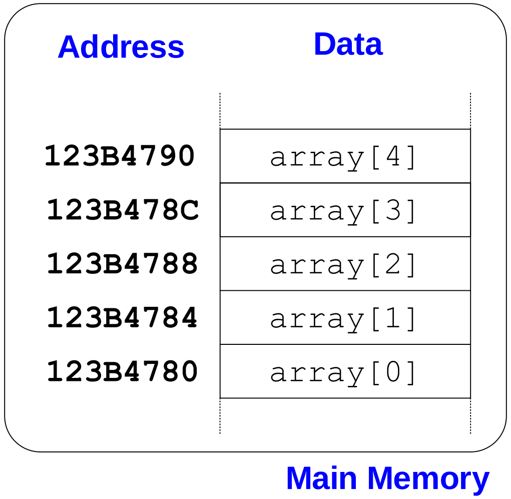
Arrays

Arrays

- Access large amounts of similar data
- **Index:** access each element
- **Size:** number of elements

Arrays

- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register

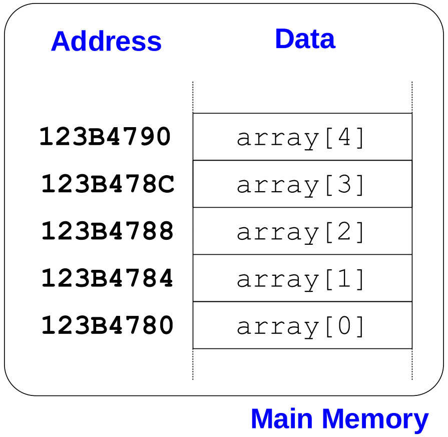


Address	Data
123B4790	array[4]
123B478C	array[3]
123B4788	array[2]
123B4784	array[1]
123B4780	array[0]

Main Memory

Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;

# RISC-V assembly code
# s0 = array base address
```



Address	Data
123B4790	array[4]
123B478C	array[3]
123B4788	array[2]
123B4784	array[1]
123B4780	array[0]

Main Memory

Accessing Arrays Using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
        array[i] = array[i] * 8;

# RISC-V assembly code
# s0 = array base address, s1 = i
```

Accessing Arrays Using For Loops

```
# RISC-V assembly code
# s0 = array base address, s1 = i
# initialization code
  lui  s0, 0x23B8F           # s0 = 0x23B8F000
  ori  s0, s0, 0x400         # s0 = 0x23B8F400
  addi s1, zero, 0           # i = 0
  addi t2, zero, 1000        # t2 = 1000

loop:
  bge  s1, t2, done          # if not then done
  slli t0, s1, 2             # t0 = i * 4 (byte offset)
  add  t0, t0, s0            # address of array[i]
  lw   t1, 0(t0)             # t1 = array[i]
  slli t1, t1, 3             # t1 = array[i] * 8
  sw   t1, 0(t0)             # array[i] = array[i] * 8
  addi s1, s1, 1             # i = i + 1
  j    loop                  # repeat
done:
```

ASCII Code

- **ASCII:** *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters: ASCII Encodings

#	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		

Accessing Arrays of Characters

```
// C Code
   char str[80] = “CAT”;
   int len = 0;

   // compute length of string
   while (str[len]) len++;

# RISC-V assembly code
# s0 = array base address, s1 = len
```

Function Calls

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

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

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

Simple Function Call

C Code

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

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  simple      # call
0x00000304         add  s0, s1, s2
...                ...

0x0000051c simple: jr   ra          # return
```

`void` means that `simple` doesn't return a value

`jal simple:`

`ra` = PC + 4 (0x00000304)

jumps to `simple` label (PC = 0x0000051c)

`jr ra:`

PC = `ra` (0x00000304)

Function Calling Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

RISC-V Function Calling Conventions

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

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
```

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 *stack* to temporarily store registers

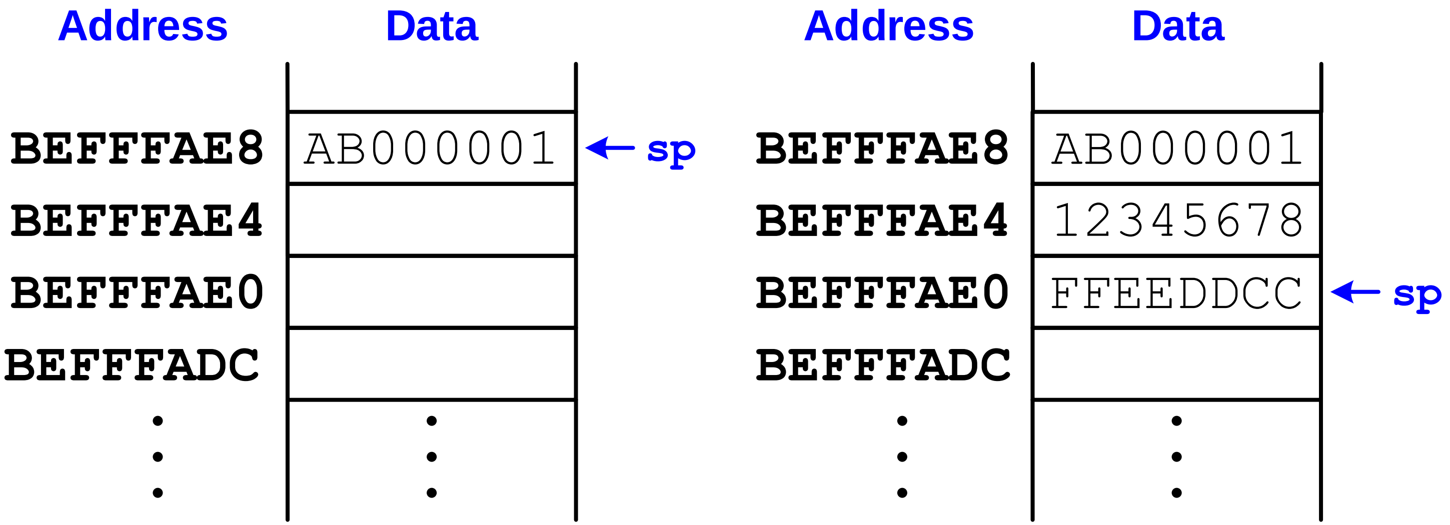
The Stack

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space is no longer needed

The Stack

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



Address	Data	
BEFFFAE8	AB000001	← sp
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
⋮	⋮	

Address	Data	
BEFFFAE8	AB000001	
BEFFFAE4	12345678	
BEFFFAE0	FFEEDDCC	← sp
BEFFFADC		
⋮	⋮	

Make room on stack for **2 words**.

How Functions use the Stack

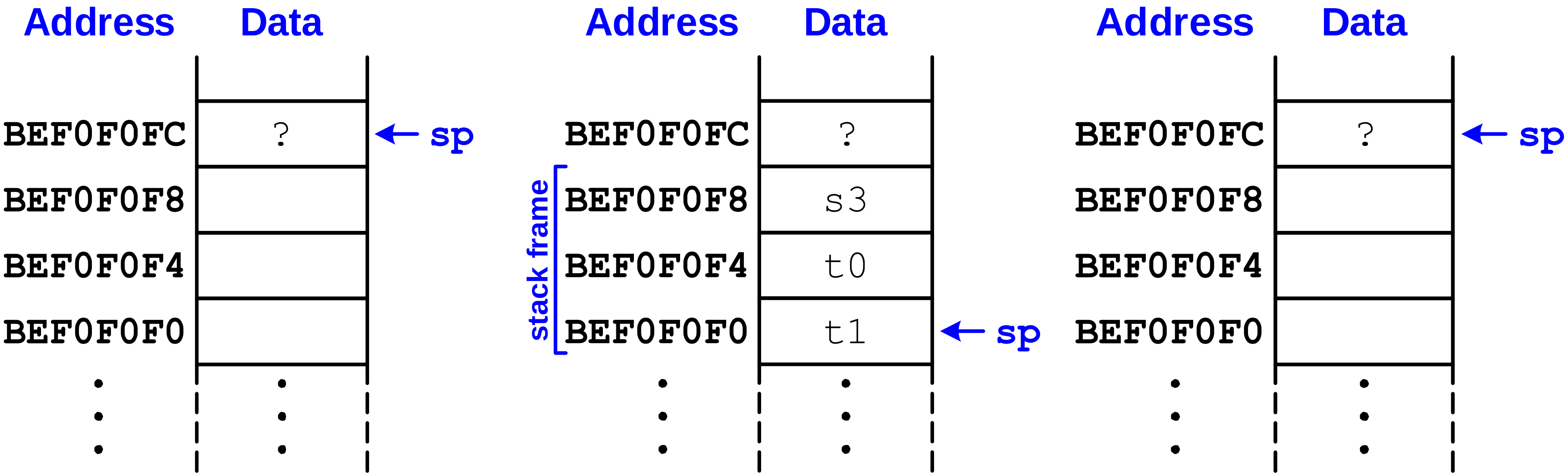
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

```
# RISC-V assembly
# 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
```

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   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
```

The Stack During `diffofsums` Call



Address	Data	
BEF0F0FC	?	← sp
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

Before Call

	Address	Data	
	BEF0F0FC	?	
stack frame	BEF0F0F8	s3	
	BEF0F0F4	t0	
	BEF0F0F0	t1	← sp
	⋮	⋮	

During Call

Address	Data	
BEF0F0FC	?	← sp
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

After Call

Preserved Registers

Preserved ***Callee-Saved***	**Nonpreserved** ***Caller-Saved***
`s0-s11`	**`t0-t6`**
`sp`	**`a0-a7`**
`ra`	
stack above **`sp`**	stack below **`sp`**

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

```
# 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

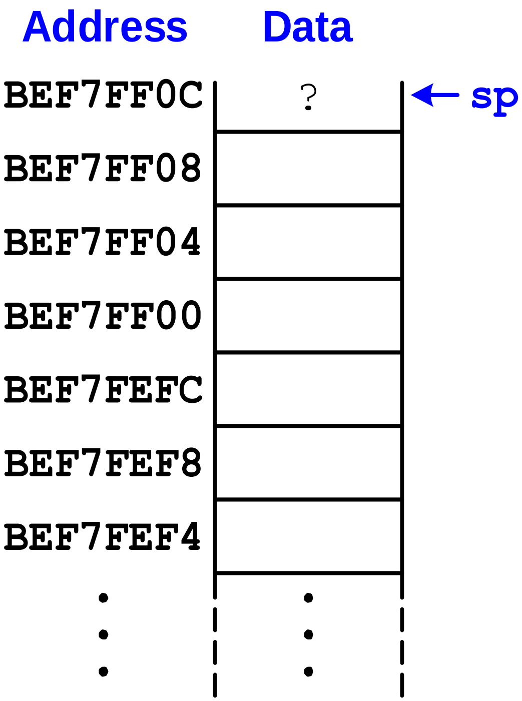
```
func1:
  addi sp, sp, -4    # make space on stack
  sw   ra, 0(sp)     # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)     # restore ra from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

Must preserve `ra` before function call.

Non-Leaf Function Call Example

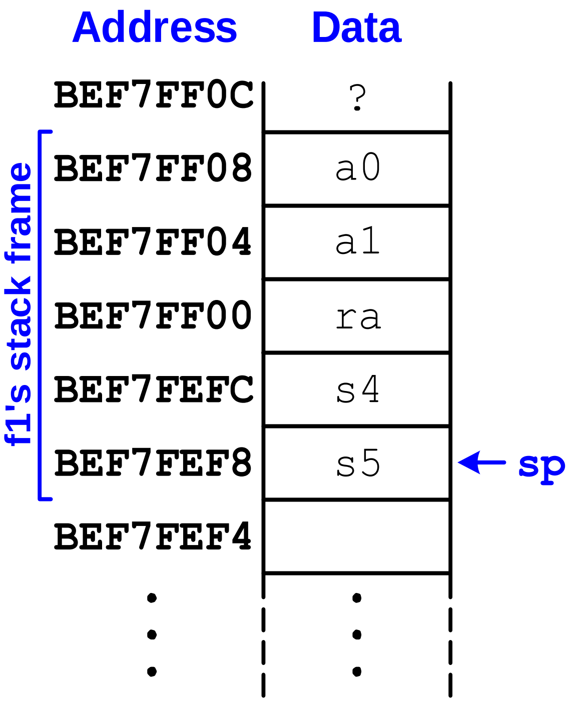
```
# f1 (non-leaf function) uses s4-s5 and needs a0-a1 after call to f2
f1:
  addi sp, sp, -20   # make space on stack for 5 words
  sw   a0, 16(sp)
  sw   a1, 12(sp)
  sw   ra, 8(sp)     # save ra on stack
  sw   s4, 4(sp)
  sw   s5, 0(sp)
  jal  func2
  ...
  lw   ra, 8(sp)     # restore ra (and other regs) from stack
  ...
  addi sp, sp, 20    # deallocate stack space
  jr   ra            # return to caller

# f2 (leaf function) only uses s4 and calls no functions
f2:
  addi sp, sp, -4    # make space on stack for 1 word
  sw   s4, 0(sp)
  ...
  lw   s4, 0(sp)
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

Stack during Function Calls



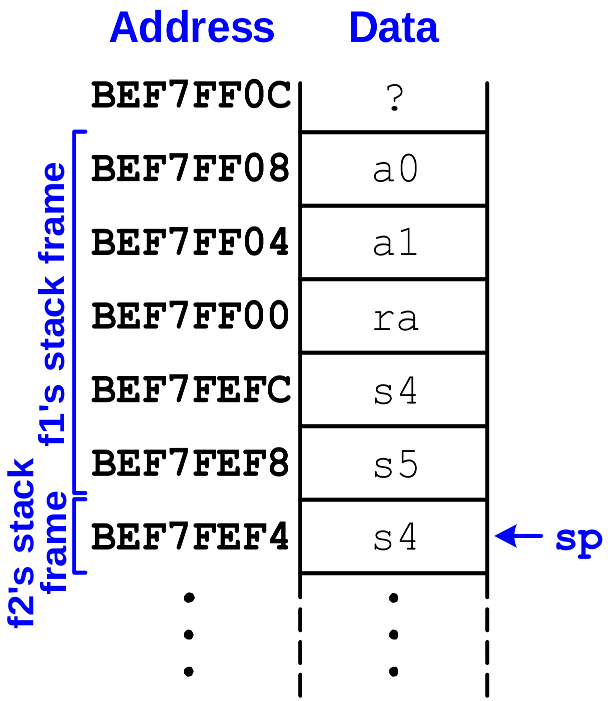
Address	Data	
BEF7FF0C	?	← sp
BEF7FF08		
BEF7FF04		
BEF7FF00		
BEF7FEFC		
BEF7FEF8		
BEF7FEF4		
⋮	⋮	

Before Calls



Address	Data	
BEF7FF0C	?	
BEF7FF08	a0	
BEF7FF04	a1	
BEF7FF00	ra	
BEF7FEFC	s4	
BEF7FEF8	s5	← sp
BEF7FEF4		
⋮	⋮	

After Call to `f1`



Address	Data	
BEF7FF0C	?	
BEF7FF08	a0	
BEF7FF04	a1	
BEF7FF00	ra	
BEF7FEFC	s4	
BEF7FEF8	s5	
BEF7FEF4	s4	← sp
⋮	⋮	

After Call to `f2`

Function Call Summary

- **Caller**
  - Save any needed registers (`ra`, maybe `t0-t6/a0-a7`)
  - Put arguments in `a0-a7`
  - Call function: `jal callee`
  - Look for result in `a0`
  - Restore any saved registers

- **Callee**
  - Save registers that might be disturbed `(s0-s11`)
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

Recursive Functions

Recursive Function Example

- Function that **calls itself**
- When converting to assembly code:
  - In the first pass, treat recursive calls as if it's calling a different function and ignore overwritten registers.
  - Then save/restore registers on stack as needed.

Recursive Function Example

- **Factorial function:**
  - factorial(n) = n!
    = n*(n-1)*(n-2)*(n-3)…*1
  - **Example:** factorial(6) = 6!
    = 6*5*4*3*2*1
    = 720

Recursive Function Example

High-Level Code

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

Example: n = 3

```
factorial(3): returns 3*factorial(2)
factorial(2): returns 2*factorial(1)
factorial(1): returns 1
```

Thus,

```
factorial(1): returns 1
factorial(2): returns 2*1 = 2
factorial(3): returns 3*2 = 6
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {




  if (n <= 1)
    return 1;



  else
    return (n*factorial(n-1));
}
```

RISC-V Assembly

```
factorial:
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

Recursive Function Example

High-Level Code

```
int factorial(int n) {




  if (n <= 1)
    return 1;


  else
    return (n*factorial(n-1));
}
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

RISC-V Assembly

```
factorial:
  addi sp, sp, -8    # save regs
  sw   a0, 4(sp)
  sw   ra, 0(sp)
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1
  addi sp, sp, 8     # restore sp
  jr   ra            # return
else:
  addi a0, a0, -1    # n = n - 1
  jal  factorial     # recursive call
  lw   t1, 4(sp)     # restore n into t1
  lw   ra, 0(sp)     # restore ra
  addi sp, sp, 8     # restore sp
  mul  a0, t1, a0    # a0=n*factorial(n-1)
  jr   ra            # return
```

Note: `n` is restored from stack into `t1` so it doesn't overwrite return value in `a0`.

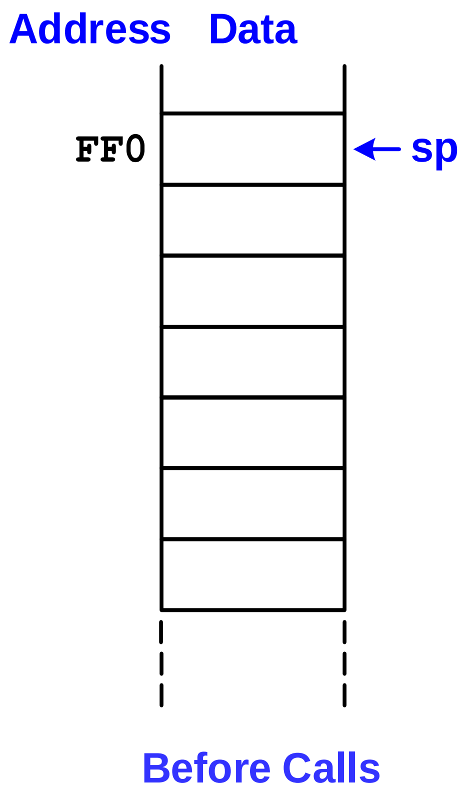
Recursive Functions

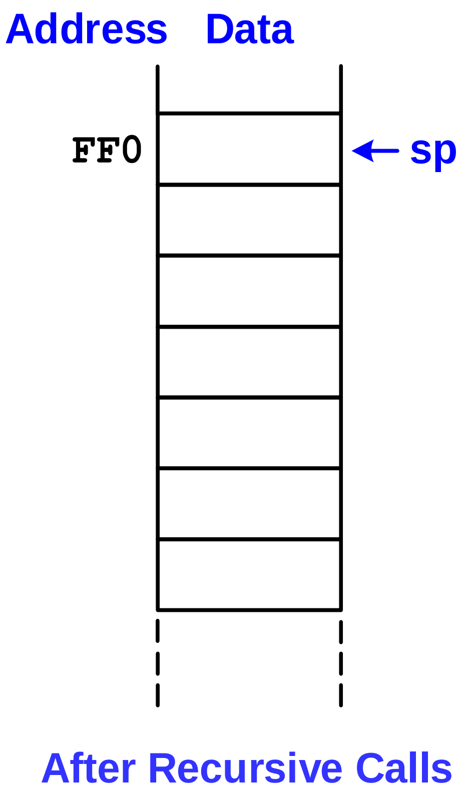
```
0x8500 factorial: addi sp, sp, -8     # save registers
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1    # temporary = 1
0x8510            bgt  a0, t0, else   # if n > 1, go to else
0x8514            addi a0, zero, 1    # otherwise, return 1
0x8518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

PC+4 = 0x8528 when factorial is called recursively.

Stack During Recursive Function

When **`factorial(3)`** is called:





More on Jumps & Pseudoinstructions

Jumps

- RISC-V has two types of unconditional jumps
  - Jump and link (`jal rd, imm`$_{20:0}$)
    - **rd** = PC+4; PC = PC + **imm**
  - jump and link register (`jalr rd, rs, imm`$_{11:0}$)
    - **rd** = PC+4; PC = [**rs**] + SignExt(**imm**)

Pseudoinstructions

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer.
- Assembler converts them to real RISC-V instructions.

Jump Pseudoinstructions

- RISC-V has four jump psuedoinstructions
  - `j    imm   jal  x0, imm`
  - `jal  imm   jal  ra, imm`
  - `jr   rs    jalr x0, rs, 0`
  - `ret        jr   ra` (i.e., `jalr x0, ra, 0`)

Labels

- Label indicates where to jump
- Represented in jump as immediate offset
  - **imm** = # bytes past jump instruction
  - In example, below, **imm** = (51C-300) = 0x21C
  - `jal simple = jal ra, 0x21C`

RISC-V assembly code

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


0x0000051c simple: jr   ra           # return
```

Long Jumps

- The immediate is limited in size
  - 20 bits for `jal`, 12 bits for `jalr`
  - Limits how far a program can jump
- Special instruction to help jumping further
  - `auipc rd, imm`: add upper immediate to PC
    - rd = PC + {$imm_{31:12}$, 12’b0}
- Pseudoinstruction: call $imm_{31:0}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset

auipc ra, $imm_{31:12}$

jalr ra, ra, $imm_{11:0}$

More RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`

`li s8, 0x56789DEF`	`lui s8, 0x5678A`
	`addi s8, s8, 0xDEF`

`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	auipc ra, $\text{imm}_{31:12}$ jalr ra, ra, $\text{imm}_{11:0}$
`ret`	`jalr zero, ra, 0`

See Appendix B for more pseudoinstructions.

Machine Language

Machine Language

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

R-Type

- ***Register-type***
- 3 register operands:
  - `rs1`, `rs2`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode*
  - `funct7`, `funct3`:

    the *function* (7 bits and 3-bits, respectively)

    with opcode, tells computer what operation to perform

R-Type

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

R-Type Examples

Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	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)**
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

More R-Type Examples

Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`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)
`srai t1, t2, 29` `srai x6, x7, 29`	32	29	7	5	6	19	0100 000	11101	00111	101	00110	001 0011	(0x41D3D313)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Machine Language:
More Formats

I-Type

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

I-Type

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

I-Type Examples

Assembly	Field Values					Machine Code					
	$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	

`addi s0, s1, 12`
`addi x8, x9, 12`
	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	`(0x00C48413)`

`addi s2, t1, -14`
`addi x18,x6, -14`
	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	`(0xFF230913)`

`lw   t2, -6(s3)`
`lw   x7, -6(x19)`
	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	`(0xFFA9A383)`

`lh   s1, 27(zero)`
`lh   x9, 27(x0)`
	27	0	1	9	3	0000 0001 1011	00000	001	01001	000 0011	`(0x01B01483)`

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

S/B-Type

- ***Store-Type***
- ***Branch-Type***
- Differ only in immediate encoding

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

S-Type

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

S-Type

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

S-Type Examples

Assembly	Field Values						Machine Code						
	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
`sw t2, -6(s3)` `sw x7, -6(x19)`	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	

B-Type

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

B-Type

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

B-Type Example

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

```
# RISC-V Assembly
0x70      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
```

imm$_{12:0}$ = 16	0	0	0	0	0	0	0	0	1	0	0	0	~~0~~
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

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

	imm$_{12,10:5}$	rs2	rs1	funct3	imm$_{4:1,11}$	op	imm$_{12,10:5}$	rs2	rs1	funct3	imm$_{4:1,11}$	op	
beq s0, t5, L1 beq x8, x30, 16	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	

U/J-Type

- ***Upper-Immediate-Type***
- ***Jump-Type***
- Differ only in immediate encoding

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

U-Type

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

U-Type

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

U-Type Example

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

Assembly

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

Field Values

imm$_{31:12}$	rd	op
0x8CDEF	21	55
20 bits	5 bits	7 bits

Machine Code

imm$_{31:12}$	rd	op
1000 1100 1101 1110 1111	10101	011 0111
20 bits	5 bits	7 bits

`(0x8CDEFAB7)`

J-Type

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

J-Type

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

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

J-Type Example

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

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

0xABC04 – 0x540C =

0XA67F8

func1 is 0xA67F8 bytes past **jal**

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**			
	imm$_{20,10:1,11,19:12}$	rd	op	imm$_{20,10:1,11,19:12}$	rd	op	

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

Review: Instruction Formats

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

Design Principle 4

Good design demands good compromises

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

Immediate Encodings

Constants / Immediates

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

C Code

```

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

RISC-V assembly code

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

Constants / Immediates

Immediate Bits

imm_{11}		$imm_{11:1}$	imm_{0}	I, S
imm_{12}		$imm_{11:1}$	0	B
$imm_{31:21}$	$imm_{20:12}$	0		U
imm_{20}	$imm_{20:12}$	$imm_{11:1}$	0	J
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 6 5 4 3 2 1	0	

Immediate Encodings

Instruction Bits

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

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

Reading

Machine Language & Addressing Operands

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 for other instruction encodings

Interpreting Machine Code

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

```
0x41FE83B3: 0100 0001 1111 1110 1000 0011 1011 0011
```

op = 51, funct3 = 0: add or sub (R-type)
funct7 = 0100000: sub

```
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 tell operation
- **Ex:** 0x41FE83B3 and 0xFDA58393

	Machine Code						Field Values						Assembly
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011	32	31	29	0	7	51	`sub x7, x29,x31` **`sub t2, t4, t6`**
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

	$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	

(0xFDA48393)	1111 1101 1010	01001	000	00111	001 0011	-38	9	0	7	19	`addi x7, x9, -38`
											`addi t2, s1, -38`

	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Addressing Modes

How do we address the operands?

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

Addressing Modes

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

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`

Example:

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

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

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

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

Assembly	imm$_{12,10:5}$	rs2	rs1	funct3	imm$_{4:1,11}$	op

`beq s8, s9, L1`
`(beq x25, x26, L1)`
	1100 101	24	25	1	0010 0	99

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

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	

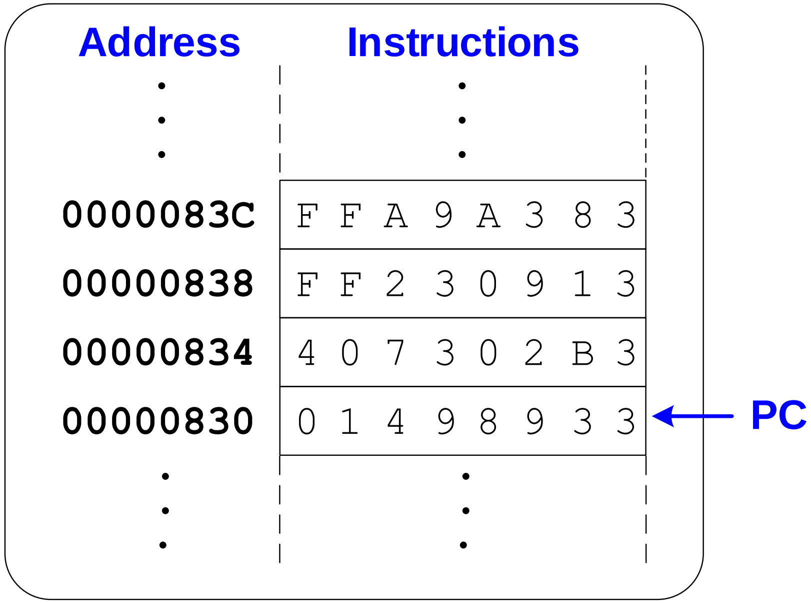
Compiling, Assembling, & Loading Programs

The Power of the Stored Program

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

The Stored Program

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

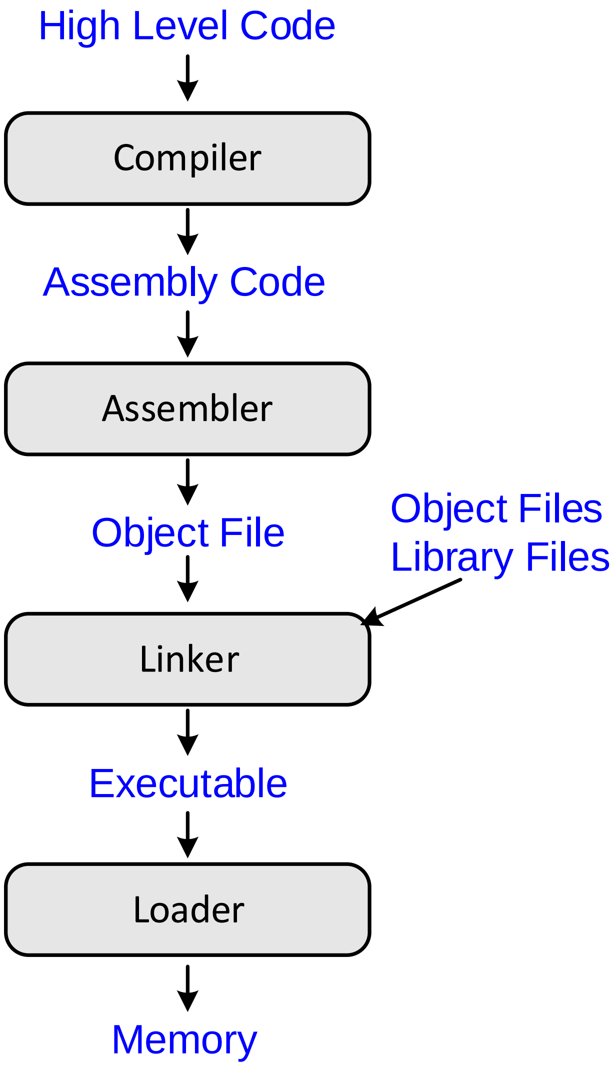


Address	Instructions
⋮	⋮
0000083C	F F A 9 A 3 8 3
00000838	F F 2 3 0 9 1 3
00000834	4 0 7 3 0 2 B 3
00000830	0 1 4 9 8 9 3 3
⋮	⋮

Main Memory

Program Counter (PC): keeps track of current instruction

How to Compile & Run a Program

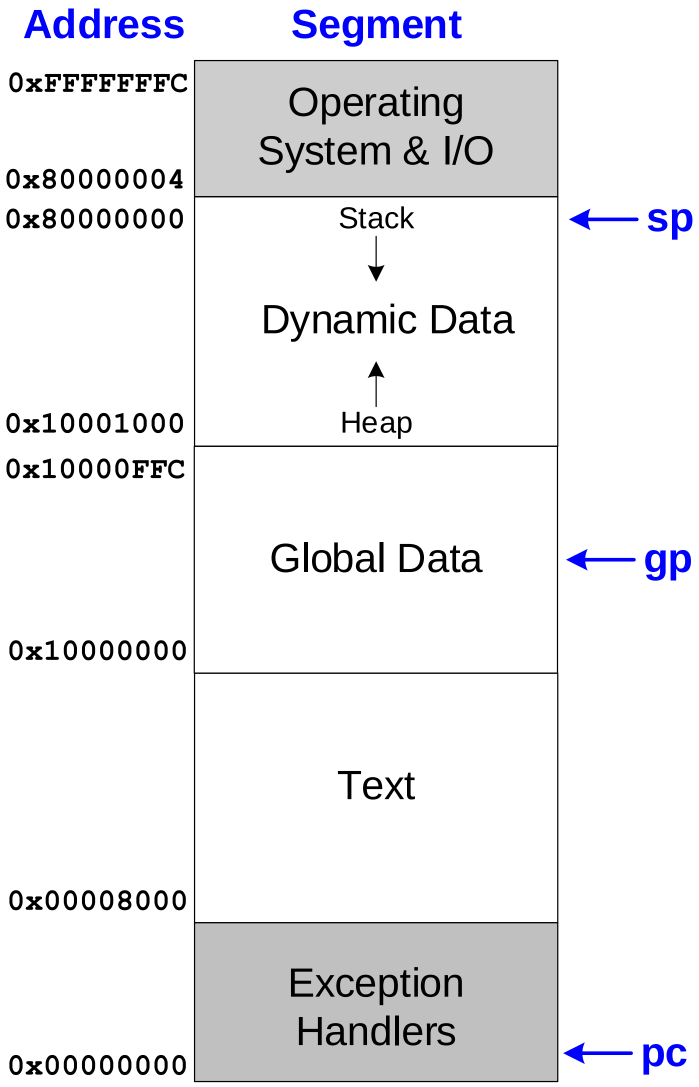


What is Stored in Memory?

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

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

Example RISC-V Memory Map



Example Program: C Code

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

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b-1));
}

void main() {
  f = 2;
  g = 3;
  y = func(f,g);

  return;
}
```

Example Program: RISC-V Assembly

```
Address  Machine Code      RISC-V Assembly Code
10144:   ff010113  func:   addi sp,sp,-16
10148:   00112623          sw   ra,12(sp)
1014c:   00812423          sw   s0,8(sp)
10150:   00050413          mv   s0,a0
10154:   00a58533          add  a0,a1,a0
10158:   0005da63          bgez a1,1016c <func+0x28>
1015c:   00c12083          lw   ra,12(sp)
10160:   00812403          lw   s0,8(sp)
10164:   01010113          addi sp,sp,16
10168:   00008067          ret
1016c:   fff58593          addi a1,a1,-1
10170:   00040513          mv   a0,s0
10174:   fd1ff0ef          jal  ra,10144 <func>
10178:   00850533          add  a0,a0,s0
1017c:   fe1ff06f          j    1015c <func+0x18>
```

Maintain **4-word alignment** of **sp** (for compatibility with RV128I) even though only space for 2 words needed.

Pseudoinstructions:

`mv`: `addi a0, s0, 0`

`ret` (return): `jr ra`

Example Program: RISC-V Assembly

Address Machine Code RISC-V Assembly Code

```
10180: ff010113 main: addi sp,sp,-16
10184: 00112623       sw   ra,12(sp)
10188: 00200713       li   a4,2
1018c: c4e1a823       sw   a4,-944(gp) # 11a30 <f>
10190: 00300713       li   a4,3
10194: c4e1aa23       sw   a4,-940(gp) # 11a34 <g>
10198: 00300593       li   a1,3
1019c: 00200513       li   a0,2
101a0: fa5ff0ef       jal  ra,10144 <func>
101a4: c4a1ac23       sw   a0,-936(gp) # 11a38 <y>
101a8: 00c12083       lw   ra,12(sp)
101ac: 01010113       addi sp,sp,16
101b0: 00008067       ret
```

`gp` = 0x11DE0

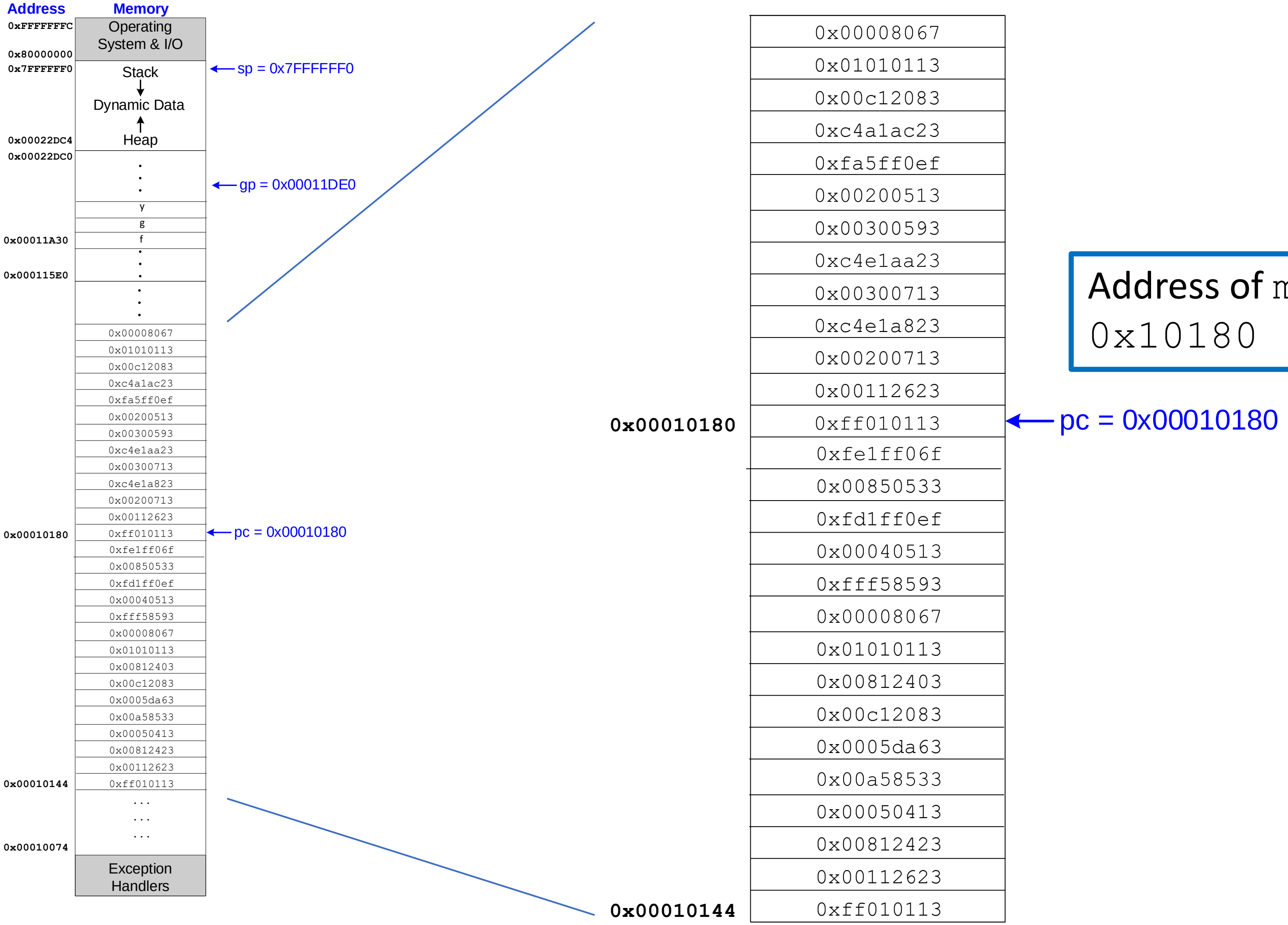
Put 2 and 3 in `f` and `g` (and argument registers) and call `func`. Then put result in `y` and return.

Example Program: Symbol Table

```
Address                         Size     Symbol Name
00010074 l      d   .text       00000000 .text
000115e0 l      d   .data       00000000 .data
00010144 g      F  .text        0000003c func
00010180 g      F  .text        00000034 main
00011a30 g      O  .bss         00000004 f
00011a34 g      O  .bss         00000004 g
00011a38 g      O  .bss         00000004 y
```

• text segment:	address 0x10074
• data segment:	address 0x115e0
• func function:	address 0x10144 (size 0x3c bytes)
• main function:	address 0x10180 (size 0x34 bytes)
• f:	address 0x11a30 (size 0x4 bytes)
• g:	address 0x11a34 (size 0x4 bytes)
• y:	address 0x11a38 (size 0x4 bytes)

Example Program in Memory



Address	Memory
0xFFFFFFFC	Operating System & I/O
0x80000000	
0x7FFFFFF0	Stack ↓
	Dynamic Data
0x00022DC4	↑ Heap
0x00022DC0	⋮
	y
	g
0x00011A30	f
0x000115E0	⋮
	⋮
	0x00008067
	0x01010113
	0x00c12083
	0xc4a1ac23
	0xfa5ff0ef
	0x00200513
	0x00300593
	0xc4e1aa23
	0x00300713
	0xc4e1a823
	0x00200713
	0x00112623
0x00010180	0xff010113
	0xfe1ff06f
	0x00850533
	0xfd1ff0ef
	0x00040513
	0xfff58593
	0x00008067
	0x01010113
	0x00812403
	0x00c12083
	0x0005da63
	0x00a58533
	0x00050413
	0x00812423
	0x00112623
0x00010144	0xff010113
	. . .
0x00010074	Exception Handlers

sp = 0x7FFFFFF0

gp = 0x00011DE0

pc = 0x00010180

	0x00008067
	0x01010113
	0x00c12083
	0xc4a1ac23
	0xfa5ff0ef
	0x00200513
	0x00300593
	0xc4e1aa23
	0x00300713
	0xc4e1a823
	0x00200713
	0x00112623
0x00010180	0xff010113
	0xfe1ff06f
	0x00850533
	0xfd1ff0ef
	0x00040513
	0xfff58593
	0x00008067
	0x01010113
	0x00812403
	0x00c12083
	0x0005da63
	0x00a58533
	0x00050413
	0x00812423
	0x00112623
0x00010144	0xff010113

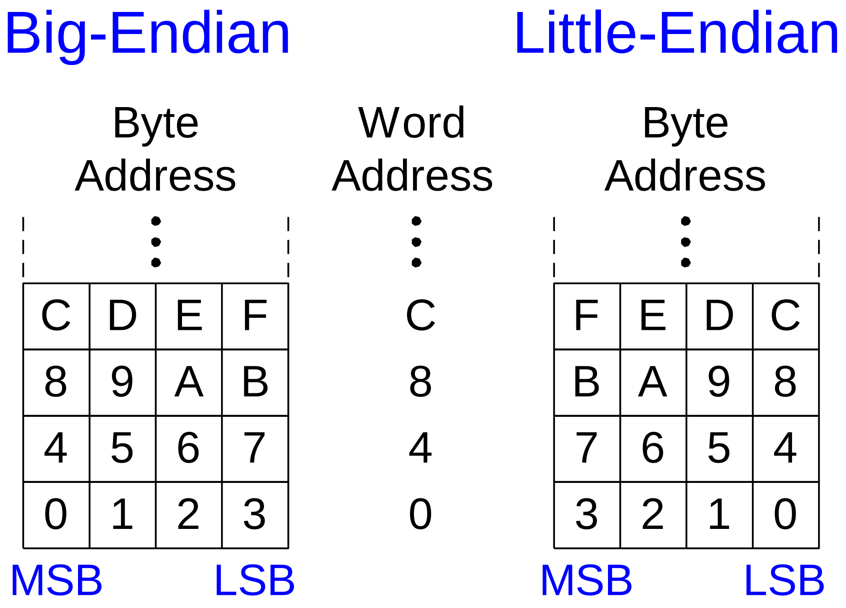
pc = 0x00010180

Address of `main`:
`0x10180`

Endianness

Big-Endian & Little-Endian Memory

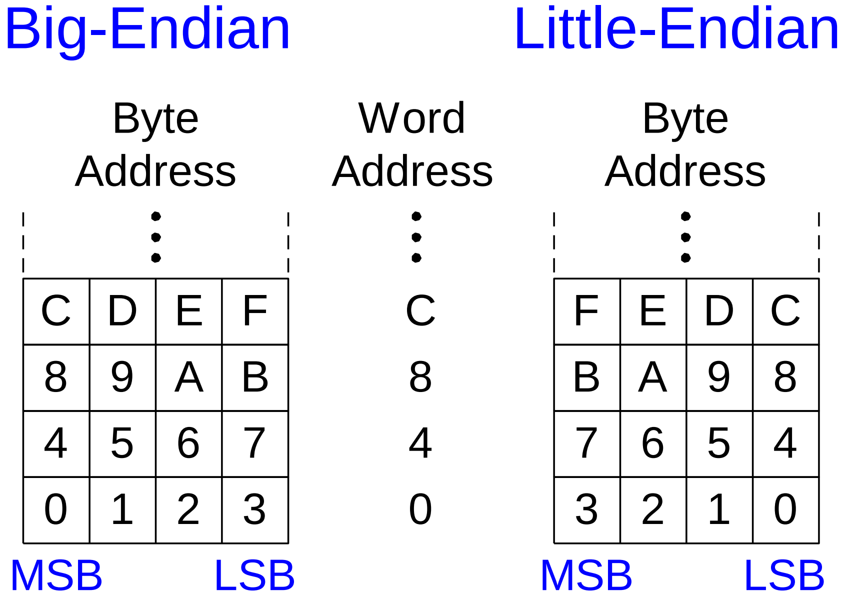
- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
	⋮			⋮		⋮		
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Memory

- Jonathan Swift’s *Gulliver’s Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn’t really matter which addressing type used – except when the two systems need to share data!



Big-Endian Little-Endian

Byte Address				Word Address	Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Example

- Suppose `t0` initially contains 0x23456789
- After following code runs on **big-endian** system, what value is `s0`?
- In a **little-endian** system?

```
sw t0, 0(zero)
lb s0, 1(zero)
```

- **Big-endian:** `s0` = 0x00000045
- **Little-endian:** `s0` = 0x00000067

Signed & Unsigned Instructions

Signed & Unsigned Instructions

- Multiplication and division
- Branches
- Set less than
- Loads
- Detecting overflow

Multiplication

- **Signed:** `mulh`
- **Unsigned:** `mulhu`, `mulhsu`
  - `mulhu`: treat both operands as unsigned
  - `mulhsu`: treat first operand as signed, second as unsigned
  - 32 lsbs are identical whether signed/unsigned; use mul

Example: s1 = 0x80000000; s2 = 0xC0000000

```
mulh s4, s1, s2
mul  s3, s1, s2
```

s1 = -2^{31}; s2 = -2^{30}

s1 x s2 = 2^{61}

s4 = 0x20000000

s3 = 0x00000000

```
mulhu s4, s1, s2
mul   s3, s1, s2
```

s1 = 2^{31}; s2 = 3×2^{30}

s1 x s2 = 3×2^{61}

s4 = 0x60000000

s3 = 0x00000000

```
mulhsu s4, s1, s2
mul    s3, s1, s2
```

s1 = -2^{31}; s2 = 3×2^{30}

s1 x s2 = -3×2^{61}

s4 = 0xA0000000

s3 = 0x00000000

Division & Remainder

- **Signed:** `div, rem`
- **Unsigned:** `divu, remu`

Branches

- **Signed:** `blt, bge`
- **Unsigned:** `bltu, bgeu`

Examples: `s1` = 0x80000000; `s2` = 0x40000000

`blt  s1, s2`
`s1` = -2^{31}; s2 = 2^{30}
`taken`

`bltu s1, s2`
`s1` = 2^{31}; s2 = 2^{30}
`not taken`

Set Less Than

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

Note: RISC-V always **sign-extends** the immediate, even for `sltiu`

Examples: `s1` = 0x80000000; `s2` = 0x40000000

```
slt  t0, s1, s2
```

`s1` = -2^{31}; s2 = 2^{30}

`t0` = 1

```
slti  t2, s1, -1   # -1 = 0xFFF
```

`s1` = -2^{31}; imm = 0xFFFFFFFF = -1

`t2` = 1

```
sltu t1, s1, s2
```

`s1` = 2^{31}; s2 = 2^{30}

`t1` = 0

```
sltiu t3, s1, -1   # -1 = 0xFFF
```

`s1` = 2^{31}; imm = 0xFFFFFFFF = 2^{32} - 1

`t3` = 1

Loads

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

Detecting Overflow

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

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

- **Example: Detecting signed overflow:**

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

Compressed Instructions

Compressed Instructions

- **16-bit** RISC-V instructions
- Replace common integer and floating-point instructions with 16-bit versions.
- Most RISC-V compilers/processors can use a **mix** of 32-bit and 16-bit instructions (and use 16-bit instructions whenever possible).
- Uses prefix: **c.**
- **Examples:**
  - `add  → c.add`
  - `lw   → c.lw`
  - `addi → c.addi`

Compressed Instructions Example

C Code

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



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

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

RISC-V assembly code

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

  c.li s1, 0              # i = 0
  addi t2, zero, 200      # t2 = 200

for:
  bge     s1, t2, done    # I >= 200? done
  c.lw    a3, 0(s0)       # a3 = scores[i]
  c.addi  a3, 10          # a3 = scores[i]+10
  c.sw    a3, 0(s0)       # scores[i] = a3
  c.addi  s0, 4           # next element
  c.addi  s1, 1           # i = i+1
  c.j     for             # repeat
done:
```

- 200 is too big to fit in compressed immediate, so noncompressed `addi` used instead.
- **`c.addi s0,4`** is equivalent to **`addi s0,s0,4`**.
- `c.bge` doesn't exist, so `bge` is used.

Compressed Machine Formats

- Some compressed instructions use a **3-bit register code** (instead of 5-bit). These specify registers `x8` to `x15`.
- **Immediates** are 6-11 bits.
- **Opcode** is 2 bits.

Compressed Machine Formats

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
funct4				rd/rs1					rs2					op		**CR-Type**
funct3			imm	rd/rs1					imm					op		**CI-Type**
funct3			imm			rs1'			imm		rs2'			op		**CS-Type**
funct6						rd'/rs1'			funct2		rs2'			op		**CS’-Type**
funct3			imm			rs1'			imm					op		**CB-Type**
funct3			imm	funct		rd'/rs1'			imm					op		**CB’-Type**
funct3			imm											op		**CJ-Type**
funct3			imm						rs2					op		**CSS-Type**
funct3			imm								rd'			op		**CIW-Type**
funct3			imm			rs1'			imm		rd'			op		**CL-Type**

Floating-Point Instructions

RISC-V Floating-Point Extensions

- RISC-V offers three floating point extensions:
  - **RVF:** single-precision (32-bit)
    - 8 exponent bits, 23 fraction bits
  - **RVD:** double-precision (64-bit)
    - 11 exponent bits, 52 fraction bits
  - **RVQ:** quad-precision (128-bit)
    - 15 exponent bits, 112 fraction bits

Floating-Point Registers

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

Floating-Point Registers

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

Floating-Point Instructions

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

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

  move (`fmv.x.w`, `fmv.w.x`)
  convert (`fcvt.w.s`, `fcvt.s.w`, etc.)
  comparison (`feq`, `flt`, `fle`)
  classify (`fclass`)
  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)

See Appendix B for additional RISC-V floating-point instructions.

Floating-Point Multiply-Add

- `fmadd` is the most critical instruction for signal processing programs.
- Requires four registers.

```
fmadd.f f1, f2, f3, f4    # f1 = f2 x f3 + f4
```

Floating-Point Example

C Code

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

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

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

RISC-V assembly code

```
# s0 = scores base address, s1 = i
  addi s1, zero, 0          # i = 0
  addi t2, zero, 200        # t2 = 200
  addi t0, zero, 10         # ft0 = 10.0
  fcvt.s.w ft0,  t0
for:
  bge    s1, t2, done       # i>=200? done
  slli   t0, s1, 2          # t0 = i*4
  add    t0, t0, s0         # scores[i] address
  flw    ft1, 0(t0)         # ft1=scores[i]
  fadd.s ft1, ft1, ft0      # ft1=scores[i]+10
  fsw    ft1, 0(t0)         # scores[i] = t1
  addi   s1, s1, 1          # i = i+1
  j      for                # repeat
done:
```

Floating-Point Instruction Formats

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

R4-Type

31:27	26:25	24:20	19:15	14:12	11:7	6:0
rs3	funct2	rs2	rs1	funct3	rd	op
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Exceptions

Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g., keyboard
  - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception
  - Jumps to exception handler
  - Returns to the program

Exception Causes

Exception	Cause
Instruction address misaligned	0
Instruction access fault	1
Illegal instruction	2
Breakpoint	3
Load address misaligned	4
Load access fault	5
Store address misaligned	6
Store access fault	7
Environment call from U-Mode	8
Environment call from S-Mode	9
Environment call from M-Mode	11

RISC-V Privilege Levels

- In RISC-V, exceptions occur at various **privilege levels**.
- Privilege levels limit access to memory or certain (privileged) instructions.
- RISC-V privilege modes are (from highest to lowest):
  - **Machine** mode (bare metal)
  - **System** mode (operating system)
  - **User** mode (user program)
  - **Hypervisor** mode (to support virtual machines)
- For example, a program running in M-mode (machine mode) can access all memory or instructions – it has the highest privilege level.

Exception Registers

- Each privilege level has registers to handle exceptions
- These registers are called control and status registers (**CSRRs**)
- We discuss **M-mode** (machine mode) exceptions, but other modes are similar
- M-mode registers used to handle exceptions are:
  - `mtvec`, `mcause`, `mepc`, `mscratch`

(Likewise, S-mode exception registers are: `stvec`, `scause`, `sepc`, and `mscratch`; and so on for the other modes.)

Exception Registers

- CSRRs are not part of register file
- M-mode CSRRs used to handle exceptions
  - **`mtvec`**: holds address of exception handler code
  - **`mcause`**: Records cause of exception
  - **`mepc`** (Exception PC): Records PC where exception occurred
  - **`mscratch`**: scratch space in memory for exception handlers

Exception-Related Instructions

- Called **privileged instructions** (because they access CSRRs)
  - **csrr**: CSR register read
  - **csrw**: CSR register write
  - **csrrw**: CSR register read/write
  - **mret**: returns to address held in mepc
- **Examples:**

```
csrr t1, mcause        # t1 = mcause
csrw mepc, t2          # mepc = t2
cwrrw t0, mscratch, t1 # t0 = mscratch
                       # mscratch = t1
```

Exception Handler Summary

- When a processor **detects an exception**:
  - It **jumps to exception handler** address in `mtvec`
  - The exception handler then:
    - **saves registers** on small stack pointed to by `mscratch`
    - Uses `csrr` (CSR read) to **look at cause** of exception (in `mcause`)
    - **Handles exception**
    - When finished, optionally **increments `mepc` by 4** and **restores registers** from memory
    - And then either **aborts** the program or **returns to user code** (using `mret`, which returns to address held in `mepc`)

Example Exception Handler Code

- Check for **two types of exceptions**:
  - **Illegal instruction** (`mcause` = 2)
  - **Load address misaligned** (`mcause` = 4)

Example Exception Handler Code

```
# save registers that will be overwritten
  csrrw t0, mscratch, t0     # swap t0 and mscratch
  sw    t1, 0(t0)            # [mscratch]   = t1
  sw    t2, 4(t0)            # [mscratch+4] = t2
# check cause of exception
  csrr  t1, mcause           # t1=mcause
  addi  t2, x0, 2            # t2=2 (illegal instruction exception code)
illegalinstr:
  bne   t1, t2, checkother   # branch if not an illegal instruction
  csrr  t2, mepc             # t2=exception PC
  addi  t2, t2, 4            # increment exception PC
  csrw  mepc, t2             # mepc=t2
  j     done                 # restore registers and return
checkother:
  addi  t2, x0, 4            # t2=4 (load address misaligned exception code)
  bne   t1, t2, done         # branch if not a misaligned load
  j     exit                 # exit program
# restore registers and return from the exception
done:
  lw    t1, 0(t0)            # t1 = [mscratch]
  lw    t2, 4(t0)            # t2 = [mscratch+4]
  csrrw t0, mscratch, t0     # swap t0 and mscratch
  mret                       # return to program
exit:
  ...
```

Checks for two types of exceptions:

- **Illegal instruction** (`mcause` = 2)
- **Load address misaligned** (`mcause` = 4)