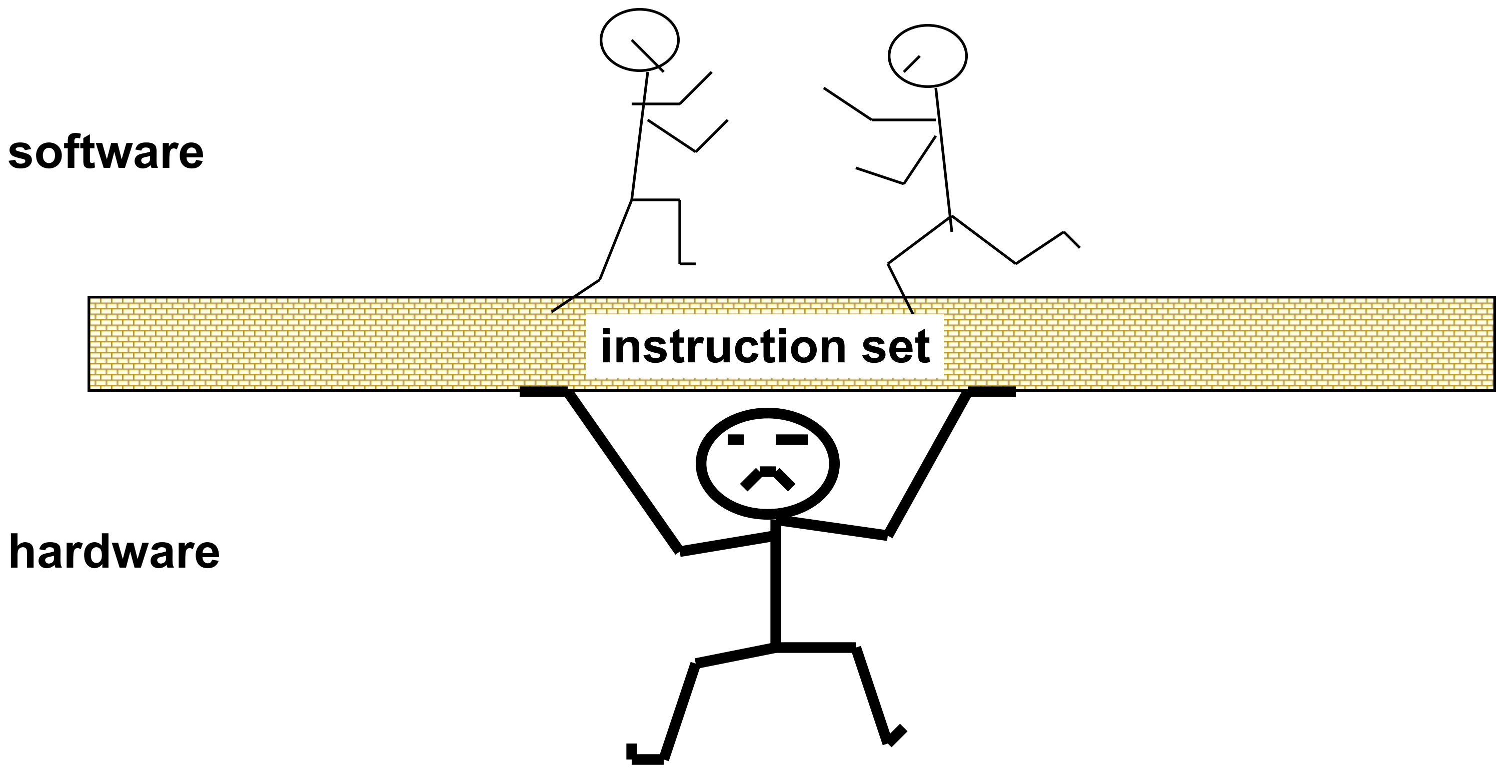
Lecture 3

- **RISC-V: Instruction Set Architecture**

Instruction Set Architecture



Instruction set provides an layer of abstraction to programmers

Levels of Representation

High Level Language Program

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

Compiler

Assembly Language Program

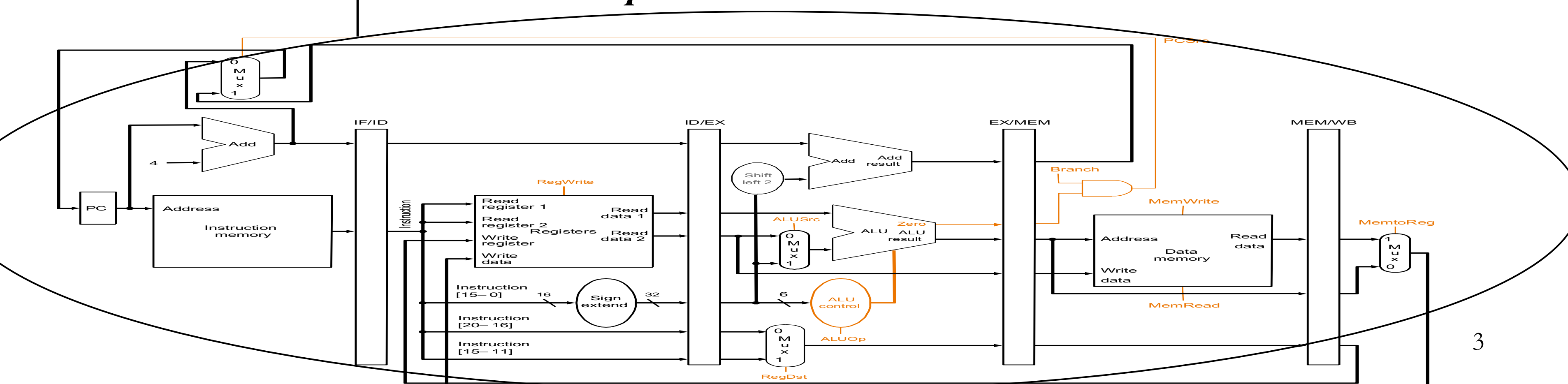
```
lw  $15,   0($2)
lw  $16,   4($2)
sw  $16,   0($2)
sw  $15,   4($2)
```

Assembler

Machine Language Program

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

Machine Interpretation



ISA Design Principle

To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost.

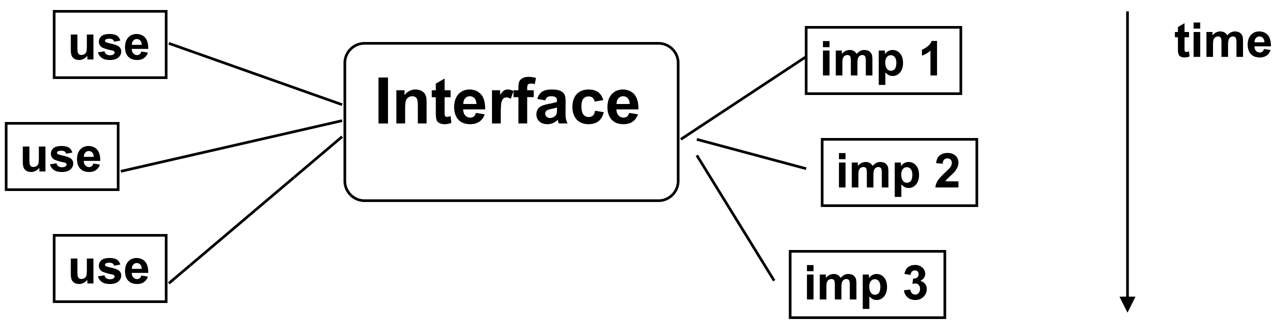
> *"It is easy to see by formal-logical methods that there exist certain [instruction set] that are in abstract adequate to control and cause the execution of any sequence of operations….The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature:* ***simplicity of the equipment*** *demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems."*

Burks, Goldstine, and von Neumann, 1947

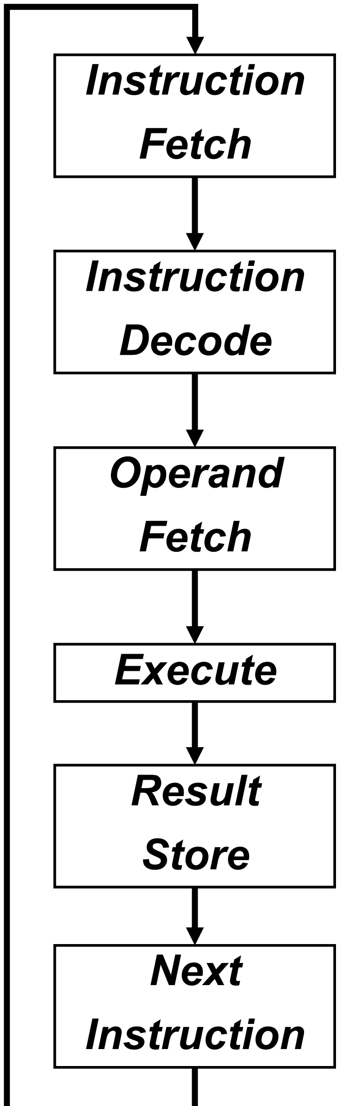
Interface Design

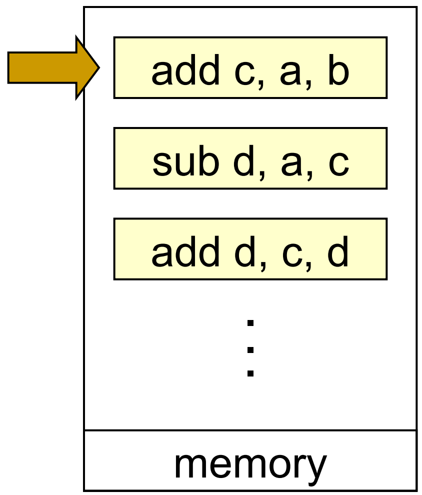
A good interface:

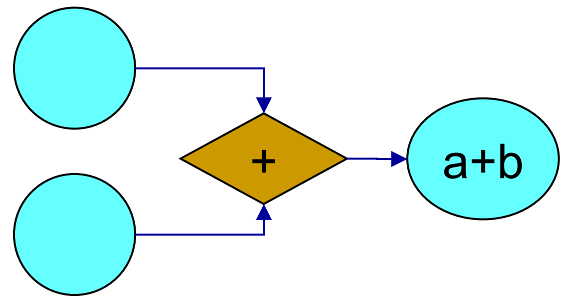
- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels



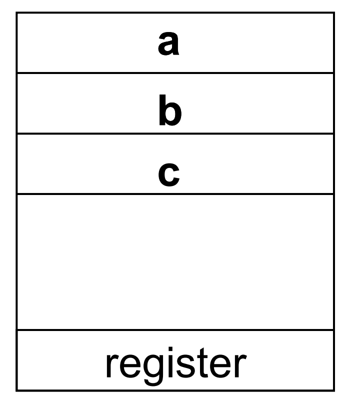
Instruction Set Architecture: What Must be Specified?







- Instruction Format or Encoding
  - – decode machine language?
- Location of operands and results
  - – where other than memory?
  - – how many explicit operands?
  - – how are memory operands located?
  - – data type and size
- Operations
  - – what are supported?
- Successor instruction
  - – jumps, conditions, branches



General Purpose Register ISA

General Purpose Register:

- register-memory

  2 address: add R1, A R1 = R1 + mem[A]

  3 address: add R2, R1, A R2 = R1 + mem[A]

- register to register (load-store)

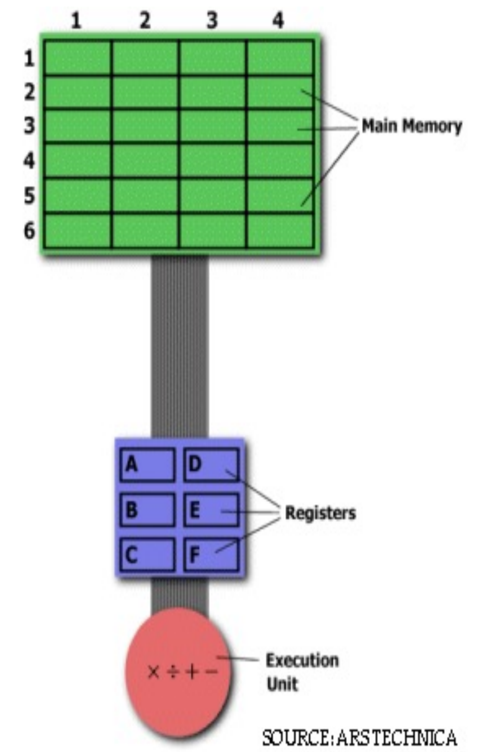
  add R2 R1 R3 R2 = R1 + R3

  load R3 A R3 = mem[A]

  store R3 A mem[A] =R3

RISC vs. CISC

- RISC (Reduced Instruction Set Architecture)
  - How to perform AxB ? (A -> 2:3 B-> 5:2)
  - LOAD A, 2:3
    LOAD B, 5:2
    MULTI A, B
    STORE 2:3, A
  - Exampel: ARM, MIPS, RISC-V
- CISC (Complex Instruction Set Architecture)
  - MULT 2:3, 5:2
  - Example: Intel x86



https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

RISC-V

- RISC-V (pronounced "risk-five") is a new instruction set architecture (ISA)
  - a standard open architecture for industry implementations.
  - RISC-V was originally developed in the Computer Science Division of the EECS Department at the University of California, Berkeley
  - Lead by Prof. David Patterson and Prof. Krstye Asonoic
- SiFive – Silicon at the speed of software
  - Founded in 2015
  - Produces computer chips based on the RISC-V instruction set architecture。

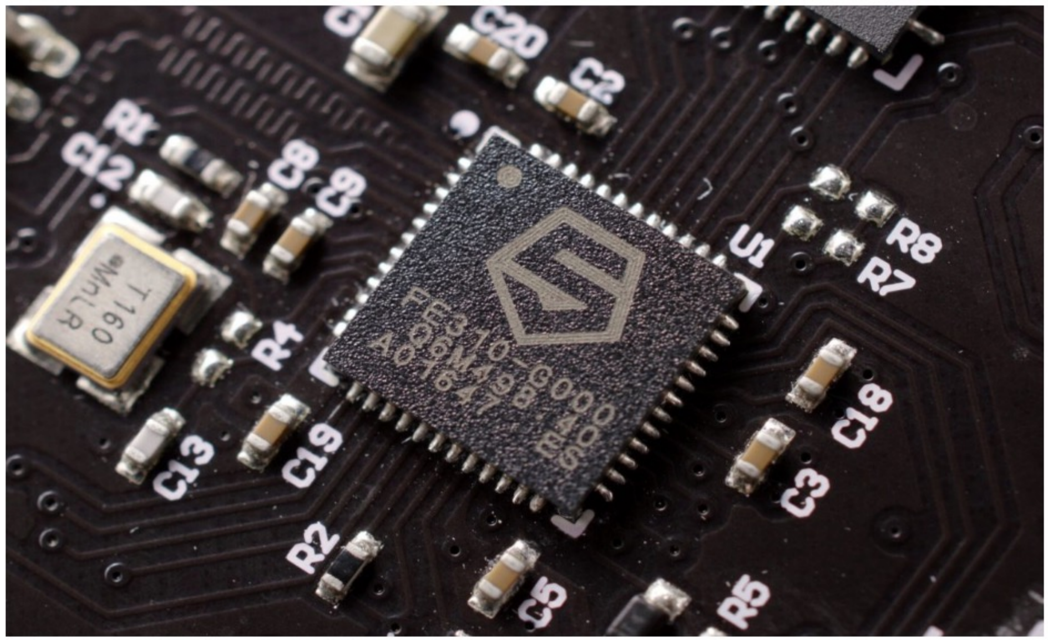
June 7, 2019

Investors are zeroing in on the open standard RISC-V instruction set architecture and the processor intellectual property being developed by a batch of high-flying chip startups.

- https://www.hpcwire.com/2019/06/07/qualcomm-invests-in-risc-v-startup-sifive/

看準中國 RISC-V 前景！SiFive：已赴中國設獨立公司衝刺

作者 MoneyDJ | 發布日期 2019 年 06 月 26 日 15:00 | 分類 國際貿易 , 晶片 分享 Follow 讚 313 分享



https://technews.tw/2019/06/26/sifive-to-china-risc-v/

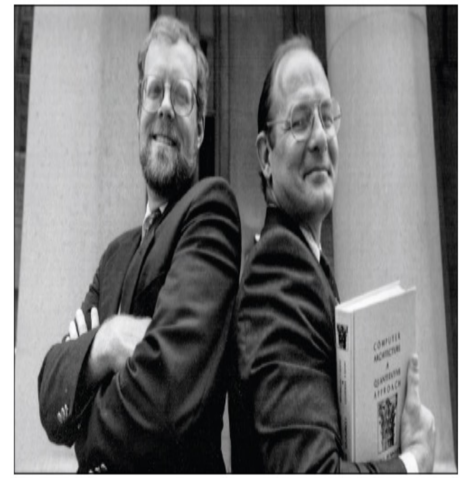
https://www.zdnet.com/article/risc-v-the-linux-of-the-chip-world-is-starting-to-produce-technological-breakthroughs/

John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018

2017 ACM A.M. Turing Award recipients John Hennessy and David Patterson delivered the Turing Lecture on June 4 at ISCA 2018 in Los Angeles. The lecture took place from 5 to 6 p.m. PDT and was open to the public. A video of the lecture can be viewed below.

Titled "A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development," the talk covers recent developments and future directions in computer architecture.

Hennessy and Patterson were recognized with the Turing Award for "pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry."



AWARDS & RECOGNITION

John Hennessy and David Patterson Receive 2017 ACM A.M. Turing Award -

https://www.youtube.com/watch?v=4OmnybxvTzA

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market (e.g., MIPS, ARMS)
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- One operation must have exactly three operands

src1 src2

Add a, b, c

Destination

- Arithmetic operations
  - +, - , x , / (more on multiply & divide later)

Design Principle 1: Simplicity favors regularity.

Arithmetic Example

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

add t0, g, h

add t1, I, j;

sub f, t0, t1;

g + h

i + j

f= ( ) + ( )

Where are the operands stored?

Register operands

- Operands of arithmetic operations must be stored in registers
  - Registers are primitive used in hardware design that are also visible to programmers
- RISC-V has a 32 $\times$ 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
    - 32 x 64-bit general purpose registers x0 to x31
  - 32-bit data is called a “word”

- Design Principle 2 : Smaller is faster

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C code:

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

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory operands

- How to load operands from memory? How to store results to memory?
  - Data transfer instructions
    - lw x9, 8 (x22) # x9 = mem[8+reg[x22]]
    - sw x9, 8 (x22) # mem[8+reg[x22]] = x9

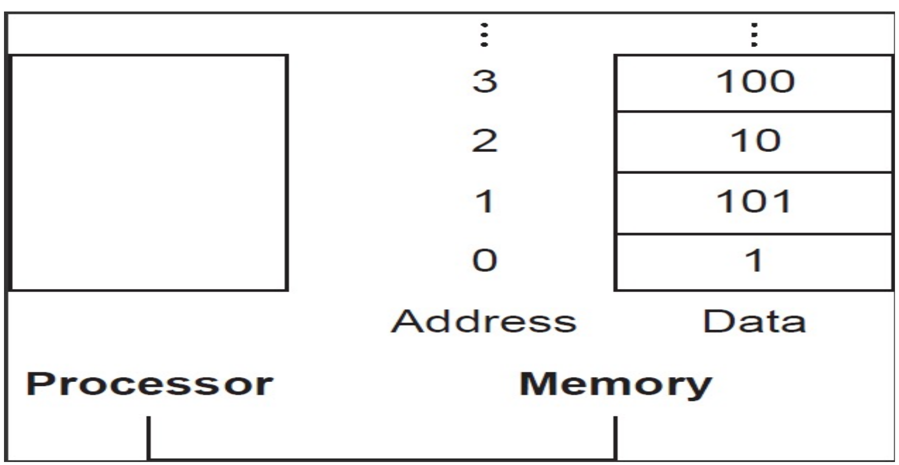
Addressing

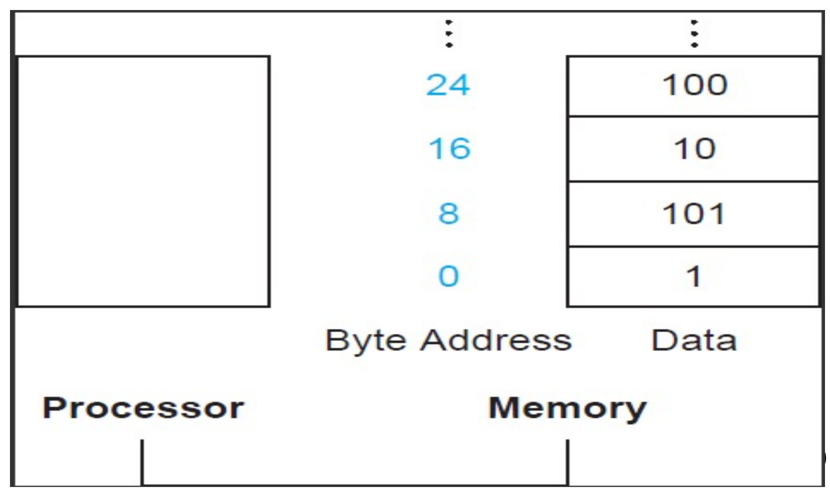
- Example: `A[12] = h + A[8];`
  - h in x21, base address of A in x22
- Compiled RISC-V code

  ld x9, offset1(x22) # x9 gets A[8] , [x22] + offset

  add x9, x21, x9

  sd x9, offset2(x22)
  - What is the offset?
    - Each element is double-word   Index * 8





ld x9, 64(x22)

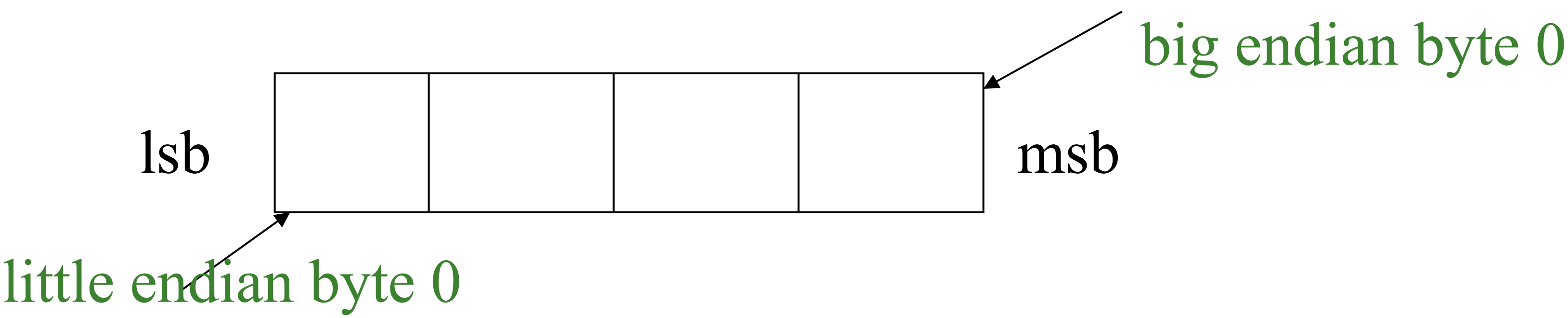
x22	100
x9	31

Addr: 100

……………………..	1	101	10	100	102	201	202	30	31	
	0	8	16	24	32	40	48	56	64	

Addressing (cont.)

- Byte order: Big Endian vs. Little Endian
  - Big endian: byte 0 is 8 most significant bits e.g., IBM/360/370, Motorola 68K, MIPS, Sparc, HP PA
  - Little endian: byte 0 is 8 least significant bits e.g., RISC-V, Intel 80x86, DEC Vax, DEC Alpha



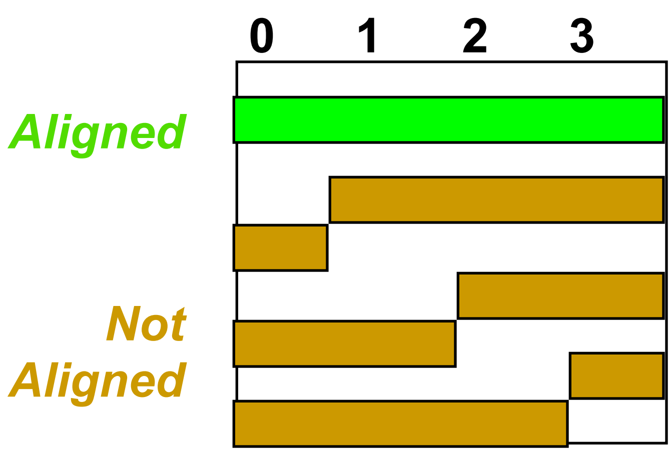
Example :

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

How is “12345678h” stored in memory?

Alignment

- RISC-V does not require that objects fall on address that is
  multiple of their size
  - Word (4 bytes): aligned if address % 4 = 0



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Constant or Immediate operands

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - Example: add constant 4 to register x22

```
ld   x21, AddrConstant4(x22)
add  x22, x22, x21
```

- RISC-V Instructions:

```
addi x22, x22, 4
```

Design Principle 3: Make the common case fast.

Representing Instruction in the Computer

add x4, x3, x2 # x4 = x3 + x2

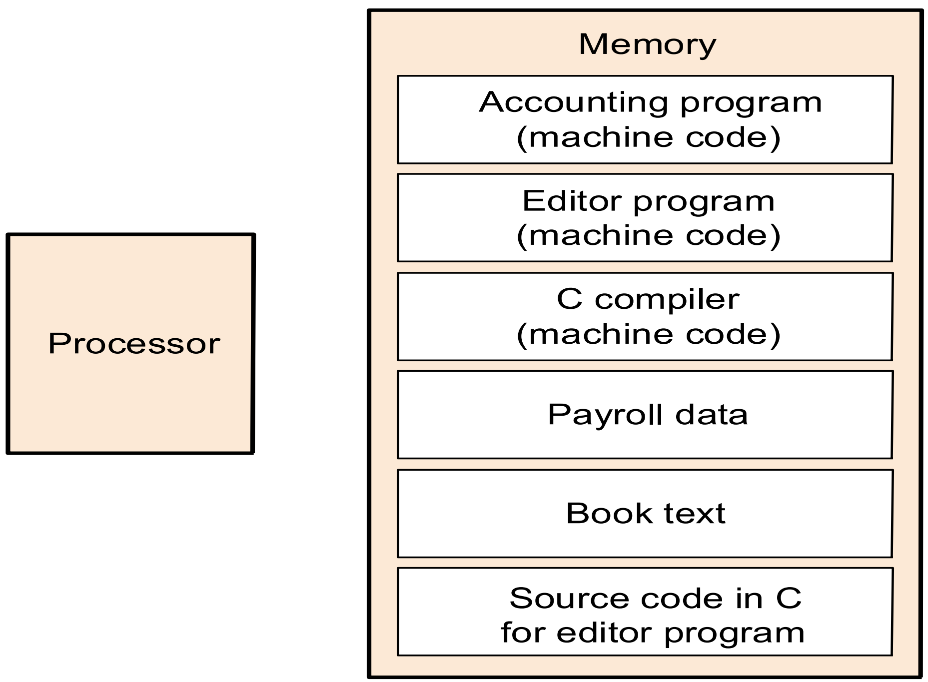
Machine language

001000	11101	11101	01000	00000	100000

All represent with binary numbers

Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Thus, entire programs can be stored in memory to be read or written just like numbers



Representing instruction/data values in Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
add x9,x20,x21
```

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} =$

$015A04B3_{16}$

Instruction format (cont.)

- Can we use the same format for lw/sw instruction?
  - 5-bit constant is too small to index arrays or data structures
  - More instruction formats

Design Principle 4 : Good design demands good compromises

RISC-V I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
  - Example : ld x9, 64(x22)

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Two's complement

00001101 (+13)
0000000000001101 (+13)

11110011 (-13)
1111111111110011 (-13)

Sign extended

RISC-V S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place
- Example : sd x9, 64(x22)

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

R-type

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-type

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

S-type

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits
    - multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits
    - divides by 2^i (unsigned only)

Logical Shift (cont'd)

- Shift right by 8 bits

  0001 0010 0011 0100 0101 0110 0111 1000

  0000 0000 0001 0010 0011 0100 0101 0110

- Shift left by 8 bits

  0001 0010 0011 0100 0101 0110 0111 1000

  0011 0100 0101 0110 0111 1000 0000 0000

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

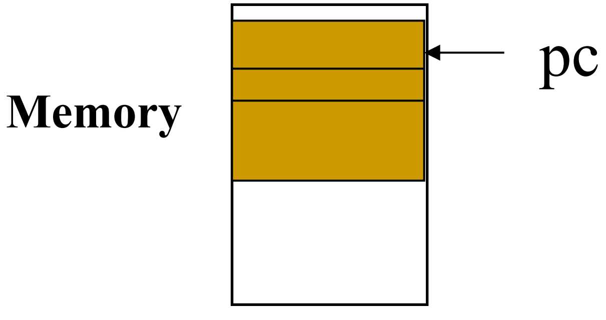
- Differencing operation
  - Set some bits to 1, leave others unchanged

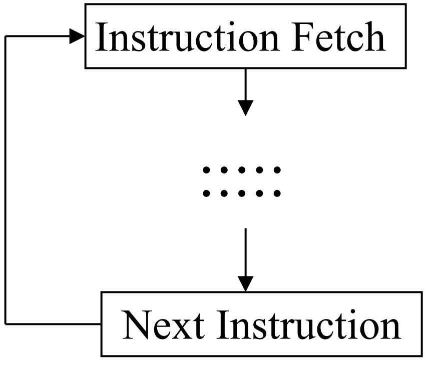
```
xor x9,x10,x12  // NOT operation
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111

Instructions for making decisions

- Decision making instructions (e.g., branch, procedure call)
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
  - I.e. change the program counter (PC)





- **Fetch instruction from mem [PC]**
- **without decision making instruction**
  - **next instruction = mem [PC + instruction_size]**

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

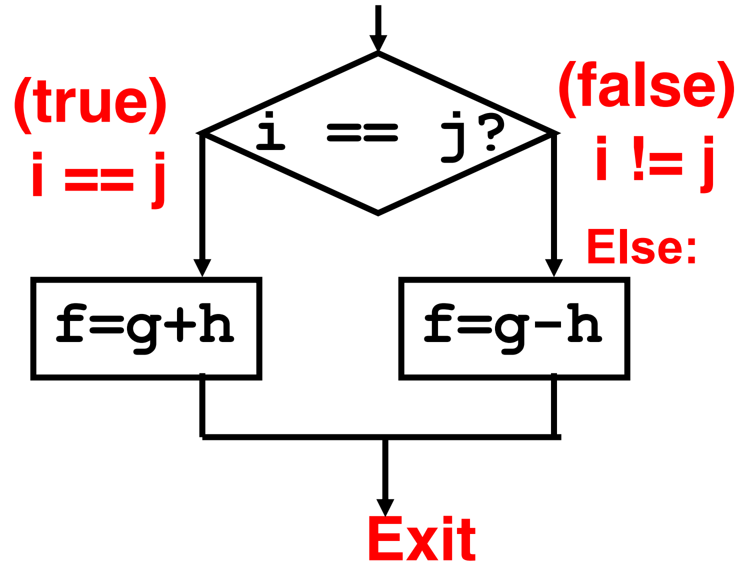
Compiling C "if" into RISC-V

```
If  (i == j)
    f = g + h;
else
    f = g – h;
```

f, g, h… in x19, x20, x21
i, j, in x22, x23…

```
    bne x22, x23, Else
    add x19, x20, x21
    beq x0,x0,Exit
Else: sub x19, x20, x21
Exit: …
```

Assembler calculates addresses



Compiling C "while" into RISC-V

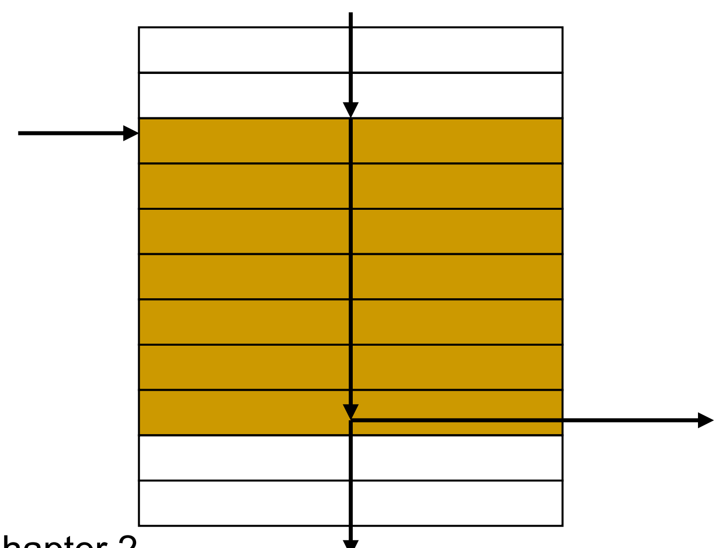
```
while (save[i] == k)
      i  += 1
```

i in x22, k in x24, address of save in x25.
each element is 8 bytes

```
 Loop: slli x10, x22, 3
       add  x10, x10, x25
       ld   x9, 0(x10)
       bne  x9, x24, Exit
       addi x22, x22, 1
       beq  x0, x0, Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- `blt rs1, rs2, L1` // branch if less than
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1` // branch if greater or equal
  - if (rs1 >= rs2) branch to instruction labeled L1
  - Example: if (a > b) a += 1; a in x22, b in x23

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

- slt reg1, reg2, reg3 //set les than

```
if (reg2 < reg3)
  reg1 = 1;                   # set
else reg1 = 0;                # reset
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1

Example

```
Switch (k)
{
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
  case 3: f = i-j; break;
}
```

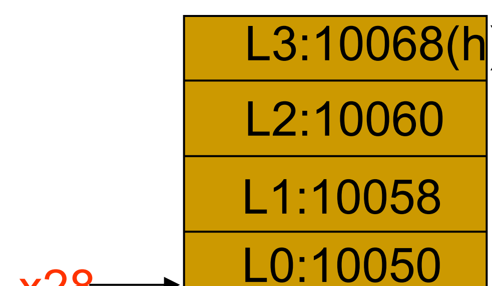
x18: k
x19: f
x20: g
x21: h
x22: i
x23: j
x5, x6, x7: temp register

```
.data:
  JumpTable: .word L0, L1, L2, L3

.text
  slt x5, x18, x0          # Test if k <0
  bne x5, x0, Exit         # if k<0, go to Exit
  slti x5, x18, 4          # Test if k<4
  beq x5, x0, Exit         # if k>=4, go to Exit
  la x28, JumpTable        # x28 = Addr of JumpTable[0]
  slli x5, x18, 2          # index
  add x6, x5, x28          # x6 = Addr of JumpTable[k]
  lw x7, 0(x6)             # x7 = JumpTable[k]
  jr x7                    # jump based on register x7

L0:   add x19, x22, x23
      j Exit
L1:   add x19, x20, x21
      j Exit
L2:   sub  x19, x20, x21
      j Exit
L3:   sub x19, x22, x23

Exit:
```



Jump address table

- Pseudo instruction
  - `la x28, JumpTable`
    - `Load address : R[x28]= Jumptable address`
  - `jr x7`
    - PC = R[x7]

Procedures

- int f1 (int i, int j, int k, int g)

```
{ ::::
 add x9, x7,x8;
  return 1;
 }

 int f2 (int s1, int s2)
 {
   ::::::
  add   x9,  x10, x11
   i = f1 (3,4,5, 6);
   add   x7, x8, x9  <-- PC
   ::::
 }
```

callee

caller

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller x10/x11
  6. Return to place of call (address in x1)

Save & restore registers

Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address

- Procedure return: jump and link register

  `jalr x0, 0(x1)`

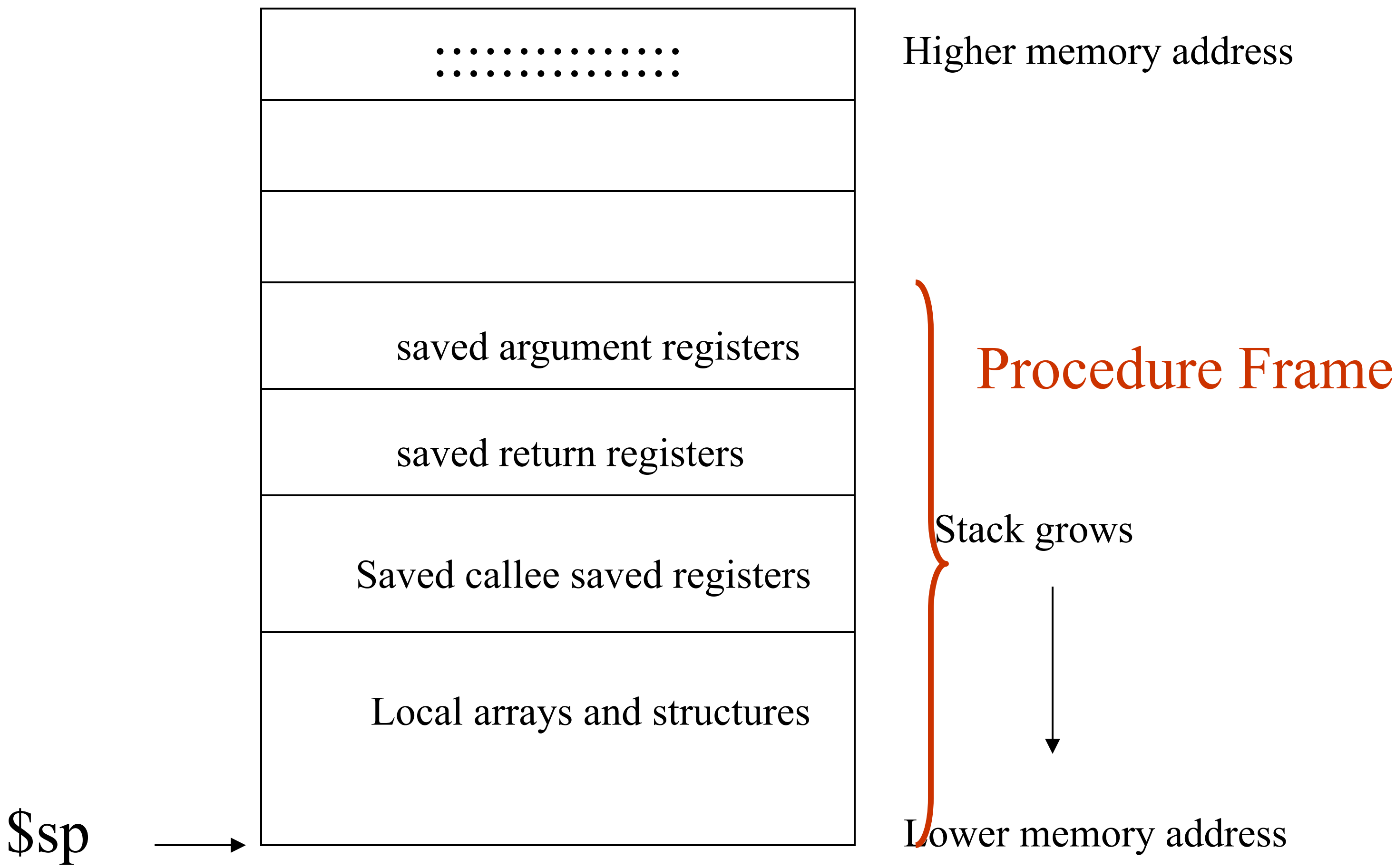
  - Like jal, but jumps to 0 + address in x1
    - PC = 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

```
int f1 (int i, int j, int k, int g
{ ::::
 add x10, x11, 1;
 return 1;
}
```

```
int f2 (int s1, int s2)
 {
  ::::::
  i = f1 (3,4,5, 6);
  add   :::::::::
  ::::
 }
```

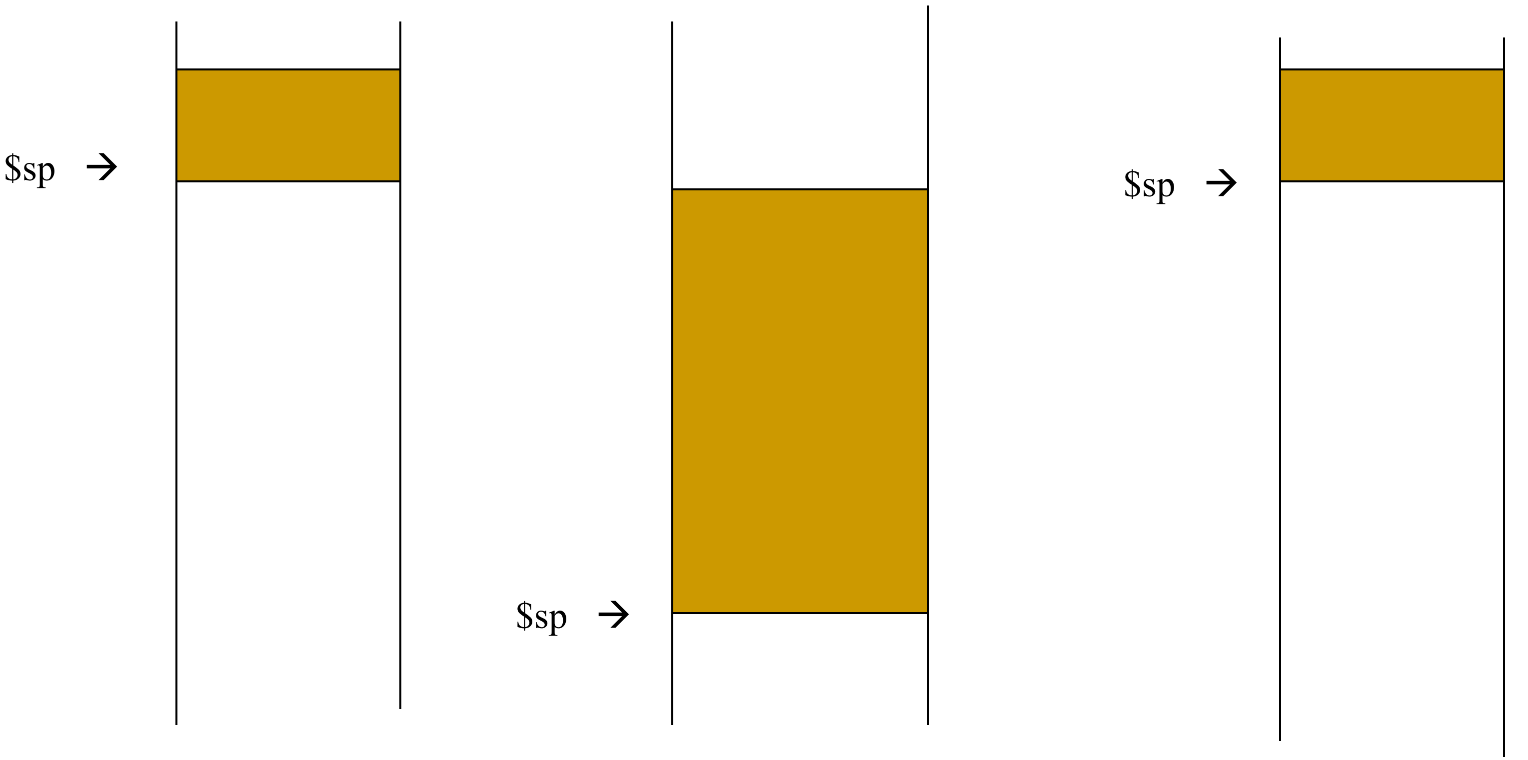
PC

Procedure Call Stack (Frame)



- stack pointer points to the top of the procedure frame

Procedure Call Stack (Frame)



Before the procedure call

during the procedure call

after the procedure call

Leaf Procedure Example

- C code:

```
long long int leaf_example (
    long long int g, long long int h,
    long long int i, long long int j)
{
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

Leaf Procedure Example

```
{
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11
  add  x6,x12,x13
  sub  x20,x5,x6
  addi x10,x20,0
  ld   x20,0(sp)
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

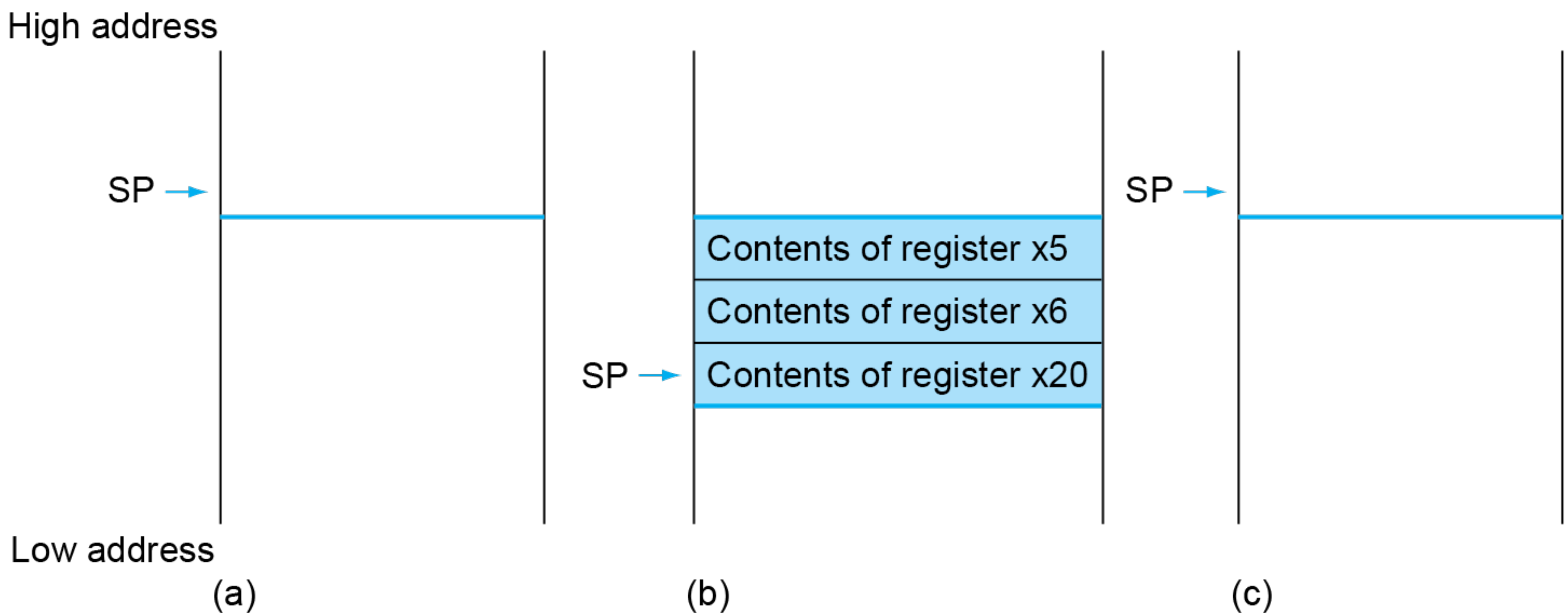
f = x5 – x6

copy f to return register

Resore x5, x6, x20 from stack

Return to caller

Local Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Leaf Procedure Example

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11
  add  x6,x12,x1
  sub  x20,x5,x6
  addi x10,x20,0
  ld   x20,0(sp)
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

f = x5 – x6

copy f to return register

Resore x5, x6, x20 from stack

Return to caller

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address: x1
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

1 x 2 x 3 x 4 ::::::::: x n

```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in x10
  - Result in x10

Non-Leaf Procedure

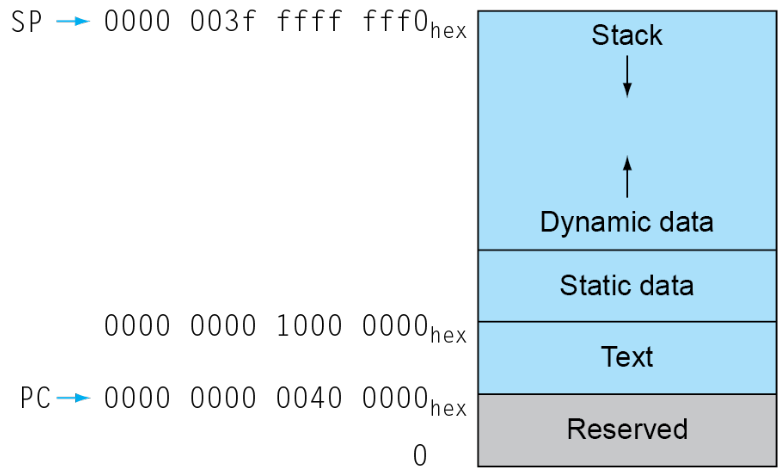
```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- RISC-V code:

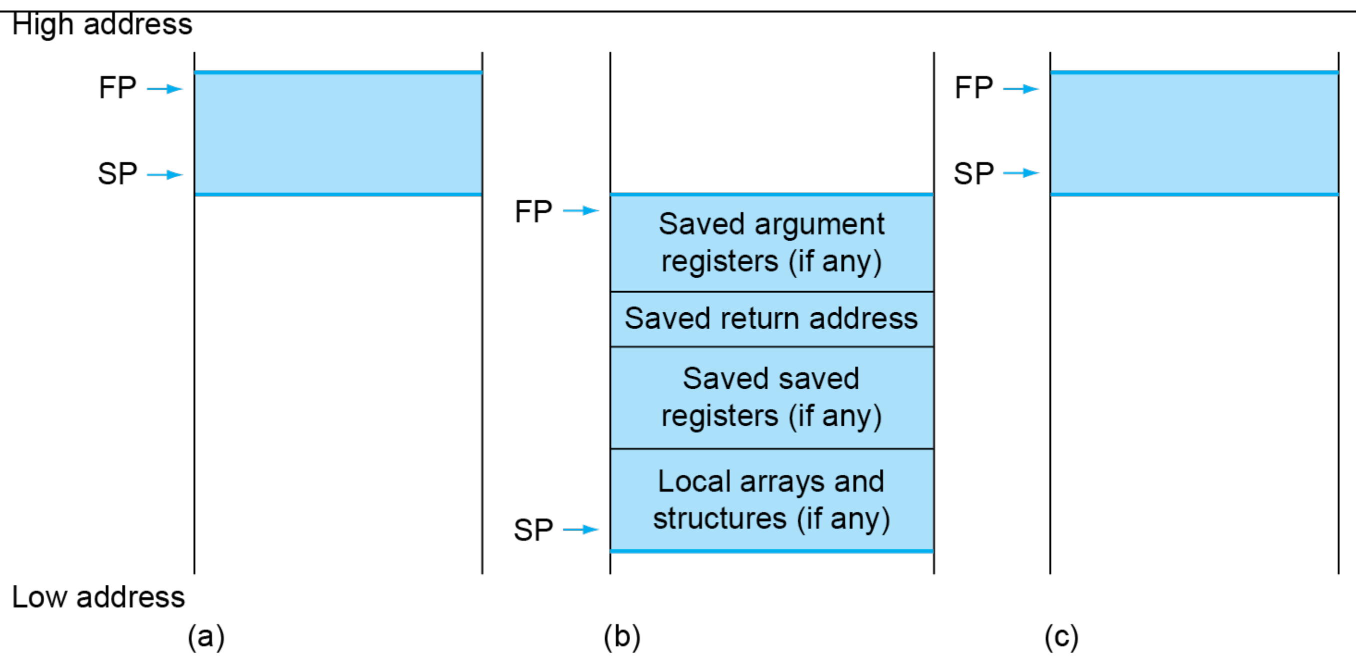
`fact:`	
`    addi sp,sp,-16`	Save return address x1 and n x10 on stack
`    sd   x1,8(sp)`	
`    sd   x10,0(sp)`	
`    addi x5,x10,-1`	x5 = n - 1
`    bge  x5,x0,L1`	if n >= 1, go to L1
`    addi x10,x0,1`	Else, set return value to 1
`    addi sp,sp,16`	Pop stack, don't bother restoring values
`    jalr x0,0(x1)`	Return
`L1: addi x10,x10,-1`	n = n - 1
`    jal  x1,fact`	call fact(n-1)
`    addi x6,x10,0`	move result of fact(n - 1) to x6
`    ld   x10,0(sp)`	Restore caller's n
`    ld   x1,8(sp)`	Restore caller's return address
`    addi sp,sp,16`	Pop stack
`    mul  x10,x10,x6`	return n * fact(n-1)
`    jalr x0,0(x1)`	return

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer saved registers
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: ~~32-bit character set~~
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

1/2/4B 2B UTF-32 → 4B

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

Arguments in X10, X11

String Copy Example

```
void strcpy (char x[], char y[])
{ size_t i;      x10          x11
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

- RISC-V code:

```
strcpy:
     addi  sp,sp,-8        // adjust stack for 1 doubleword
     sd    x19,0(sp)       // push x19
     add   x19,x0,x0       // i=0
L1:  add   x5,x19,x11      // x5 = addr of y[i]
     lbu   x6,0(x5)        // x6 = y[i]
     add   x7,x19,x10      // x7 = addr of x[i]
     sb    x6,0(x7)        // x[i] = y[i]
     beq   x6,x0,L2        // if y[i] == 0 then exit
     addi  x19,x19,1       // i = i + 1
     jal   x0,L1           // unconditional jump, next iteration of loop
L2:  ld    x19,0(sp)       // restore saved x19
     addi  sp,sp,8         // pop 1 doubleword from stack
     jalr  x0,0(x1)        // and return
```

32-bit Constants

- How to load a 32-bit constant into a register?

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

I-Type Instruction

32-bit Constants

```
lui rd, constant
```

- Copies 20-bit constant to bits [31:12] of rd
- Extends bit 31 to bits [63:32]
- Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

32-bit Constants

Example:

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,1280  // 0x500
```

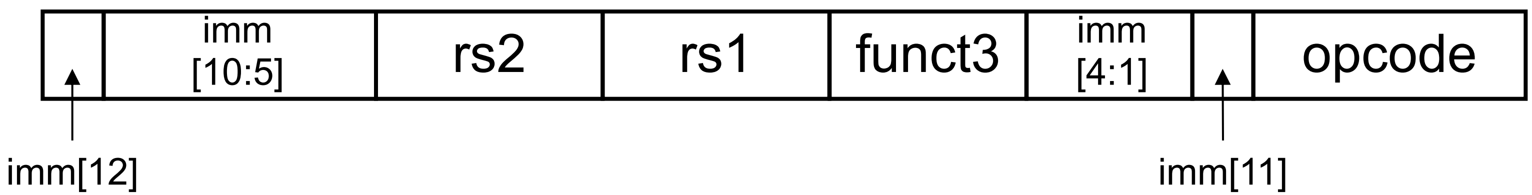
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

U Type

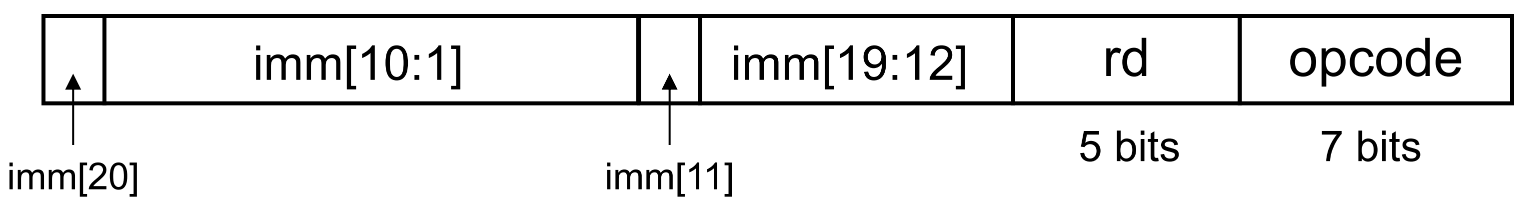
imm[31:12]	rd	opcode
20 bits	5 bits	7 bits

Encoding for Control flow instructions

- Branches – bne, beq
  - SB format



- Unconditional jump – jal, jalr
  - UJ format



Branch Addressing

- SB format:

	imm [10:5]	rs2	rs1	funct3	imm [4:1]		opcode
imm[12]						imm[11]	

Example: bne x10, x11, 2000 // if x10 !=x11, go to location 2000_{ten} = 0, 0111, 1101, 0000

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm [10:5]					imm[11]	

- PC-relative addressing
  - Target address = PC + immediate[] $\times$ 2
  - Supporting the possibility of 2-byte long instruction

Jump Addressing

- Jump and link (`jal`) target uses 20-bit immediate for larger range
- UJ format:

	imm[10:1]		imm[19:12]	rd	opcode
imm[20]		imm[11]		5 bits	7 bits

- example: jal x0, 2000 (0, 0111, 1101, 0000)

0	1111101000	0	00000000	00000	1101111
imm[20]		imm[11]		5 bits	7 bits

- PC-relative addressing

Branch Address Example

:

:

80016 bne x9, x24, Exit

inst 1;

inst 2;

Exit:

0	0				6	0	
?	?	11000	01001	001	?	0?	1100111
imm[12]	imm [10:5]					imm[11]	

	imm [10:5]	rs2	rs1	funct3	imm [4:1]		opcode
imm[12]						imm[11]	

Branching Far Away

- What if we want to branch farther than can be represented in the 12 bits of the conditional branch instruction?

```
beq   x10,  x0,   L1
```

```
        bne  x10, x0, L2
        jal     x0, L1
   L2:
```

Long Jump

- For long jumps, e.g., to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

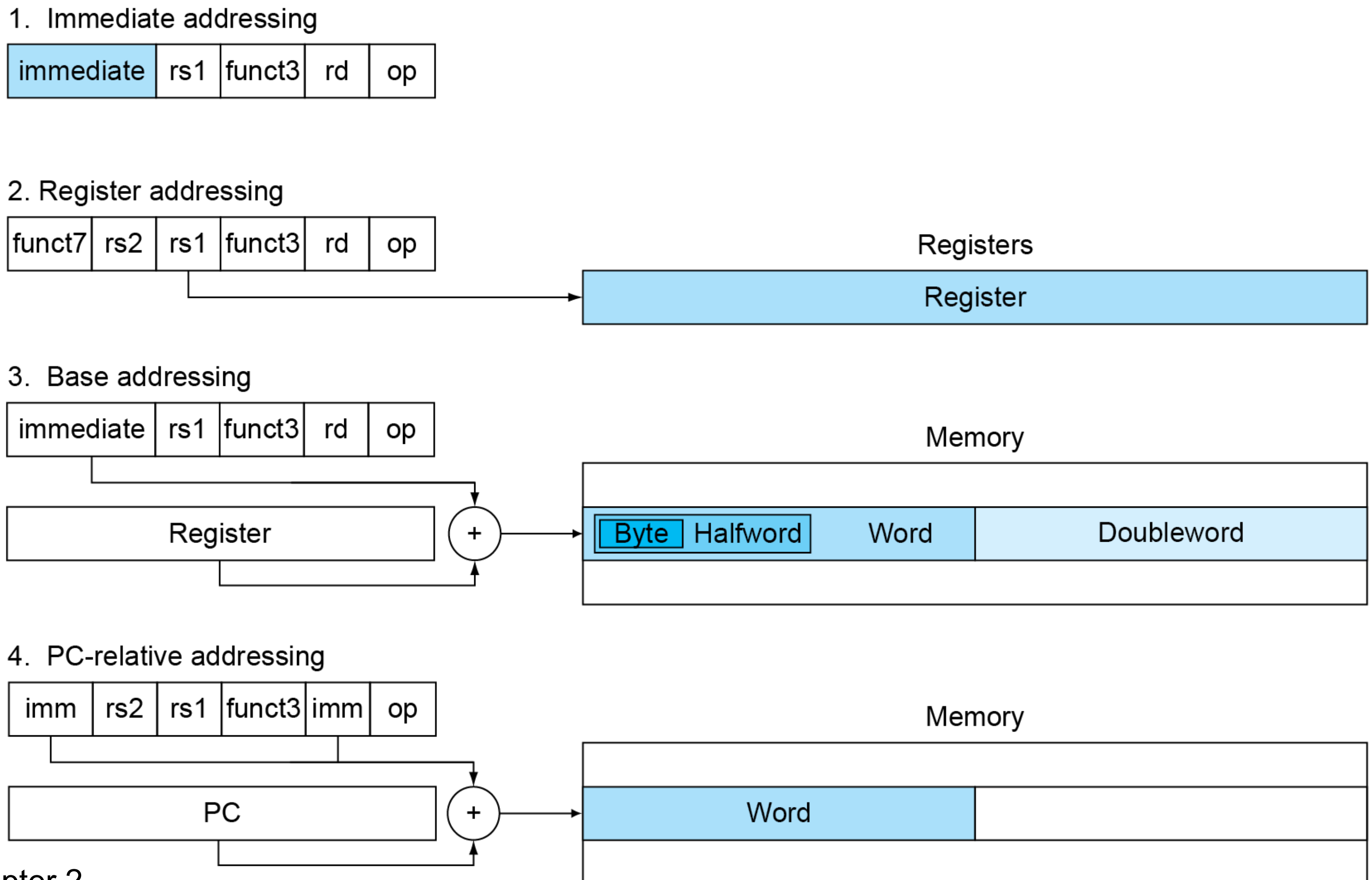
0000 0000 0011 1101 0000	0101 0000 0000

lui x8, address[31:12]

0000 0000 0011 1101 0000	0000 0000 0000

jalr x0, address[11:0](x8)

RISC-V Addressing Summary



RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Parallelism and Instructions: Synchronization

- Parallel tasks must synchronize to avoid data race, where the results of the program can change depending on how events happen to occur.
- Lock/unlock: ensure only one task entering the critical section

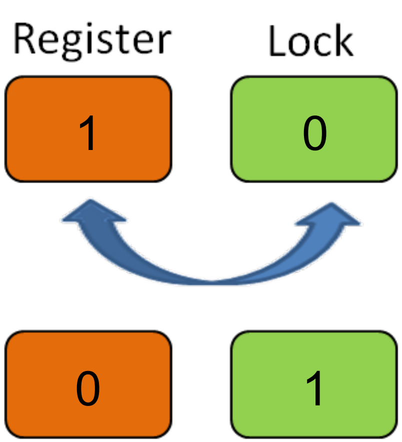
P(1)
Acquire Lock;
If Lock = 0
  enter critical section;
Release Lock;

P(2)
Acquire Lock;
If Lock = 0
  enter critical section;
Release Lock;

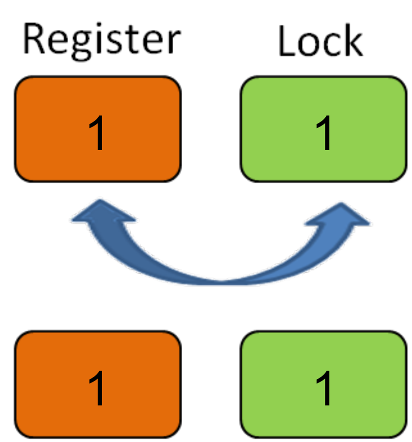
Parallelism and Instructions: Synchronization

- **Atomic SWAP**: **atomically** **interchange a value in a register for a value in memory; nothing else can interpose itself between the read and the write to the memory location**

P(1)
x4 =1;
Swap (x4, Lock);
If x4 = 0
   enter critical section;

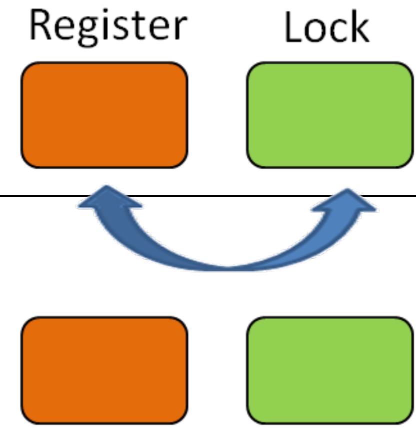


P(2)
x4 =1;
Swap (x4, Lock);
If x4 = 0
   enter critical section;



- Processor I

Processor II



Processor I:

```
        li    x4,#1
lockit: lw    x2,0(x1)
        sw    x4,0(x1)
        bnez  x2,lockit
```

Processor II:

```
        li    x4,#1
lockit: lw    x2,0(x1)
        sw    x4,0(x1)
        bnez  x2,lockit
```

initial value of lock is 0

lw x2, 0(x1) //processor 1

lw x2, 0(x1) //processor 2

sw x4, 0(x1) //processor 1

sw x4, 0(x1) //processor 2

Both processors think they get the lock

Synchronization in RISC-V

- Load reserved: `lr.d rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.d rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable, lock variable is stored at [x20], x23 initially set to 1)

```
again: lr.d x10,(x20)
       sc.d x11,(x20),x23 // X11 = status
       bne  x11,x0,again  // branch if store failed
       addi x23,x10,0     // X23 = loaded value
```

- Example 2: lock

```
       addi x12,x0,1          // copy locked value
again: lr.d x10,(x20)         // read lock
       bne  x10,x0,again      // check if it is 0 yet
       sc.d x11,(x20),x12     // attempt to storE
        bne  x11,x0,again     // branch if fails
```

- Unlock:

```
       sd   x0,0(x20)         // free lock
```