Computer Architecture

COMP SCI 2GA3

Chapter 2 - Instructions: Language of the Computer

Based on: RISC-V Chapter 2 textbook slides
COMPSCI 2GA3 2016 fall - Chapter 2
SOFTENG 2GA3 2020 winter - Chapter 2

Dr. Bojan Nokovic, P.Eng.
McMaster University, Fall Term 2021/22

Instruction Set

The **instruction set** of a computer is its repertoire of instructions that it can perform. ISA defines the interface between hardware and software.

Different computers have different instruction set, but with many aspects in common.

Early computers had very simple instruction sets.

As resources expanded, we got Complex Instruction Set Computers (CISC).

Many modern computers now also have simple instruction sets called Reduced Instruction Set Computers (RISC).

The RISC-V Instruction Set

- Developed at UC Berkeley starting in 2010 as open ISA, RISC-V instructions are 32 bits [31:0].

- Now managed by the RISC-V Foundation (riscv.org)

- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
  - Base ISAs: RV32I, RV64I, RV128I

- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

- For embedded applications where code size is important, a 16-bit instruction set exists, RISC-V compressed.

Arithmetic Operations

- Add and subtract, three operands, two sources and one destination.

```
add a, b, c  // store b + c in a
```

All RISC-V arithmetic operations have this form!

- Design Principle 1: Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands

Arithmetic instructions use register operands

- Use for frequently accessed data
- 64-bit data is called a “doubleword”
  - 32x64-bit general purpose registers x0 to x31
- 32-bit data is called a “word”
  - 32x32-bit general purpose registers x0 to x31
- RISC-V architects conceived multiple variants of the ISA (RV128, RV64, RV32, RV16)

Design Principle 2: Smaller is faster

https://en.wikipedia.org/wiki/RISC-V

RISC-V Registers

- x0: the constant value 0 (zero)
- x1: return address (ra)
- x2: stack pointer (sp)
- x3: global pointer (gp)
- x4: thread pointer (tp)
- x5 – x7, x28 – x31: temporaries (t0-t2, t3-t6)
- x8: frame pointer/ saved register (s0)
- x9, x18 – x27: saved registers (s1, s2-s11)
- x10 – x11: function arguments/results (a0-a1)
- x12 – x17: function arguments (a2-a7)

Register Operand Example

- C code:

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

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

- Compiled RISC-V code:

```
add x5, x20, x21 // x5 = g + h
add x6, x22, x23 // x6 = i + j
sub x19, x5, x6  // x19 = x5 - x6
```

RISC-V Simulator: https://www.kvakil.me/venus/

Endians

Computers can be divided by how they store a multibyte number (say a word) in byte-addressable memory

Consider the 32 bit hexadecimal number 90AB12CD

To store it, we would have to use 4 bytes, but do we store the left most digits first, or the right most?

Big-endian (most-significant byte at least address):
`mem[0]=90, mem[1]=AB, mem[2]=12, mem[3]=CD`

Little-endian(Least-significant byte at least address of a word):
`mem[0]=CD, mem[1]=12, mem[2]=AB, mem[3]=90`

RISC-V is Little-endian!

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
  - Little Endian (least-significant byte at least address of a word)
- RISC-V does not require words to be aligned in memory, unlike some other ISAs.

Memory Operands

Memory is essentially a large, single dimensional array

- The address acts as the index of the array
- Addresses start at zero and go to 2^{32} -1

The value of `mem[2]` is 10

Memory addresses and contents of memory at those locations - each memory element is 1 byte

Q: What would be addresses if memory elements are words?

Load Operation

- Data transfer operations that copy data from memory to a register are called load operations

- The format of the load instruction is the name of the operation followed by the register to be loaded, then register and a constant used to access memory

RISC-V load example:

```
lw x9, 4(x22)
```

Copies data stored at `mem[x22+4]` into register x9

Store Operation

Data transfer operations that copy data from a register to memory are called store operations

It copies data from a register to memory

The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then the base register, and finally the offset to select the array element.

RISC-V store example:

```
sw x9, 4(x22)
```

Copies data stored in register x9 to `mem[x22+4]`

Memory Operand RISC-V Example

A word requires 4 bytes to store it
Address of subsequent words thus differ by 4

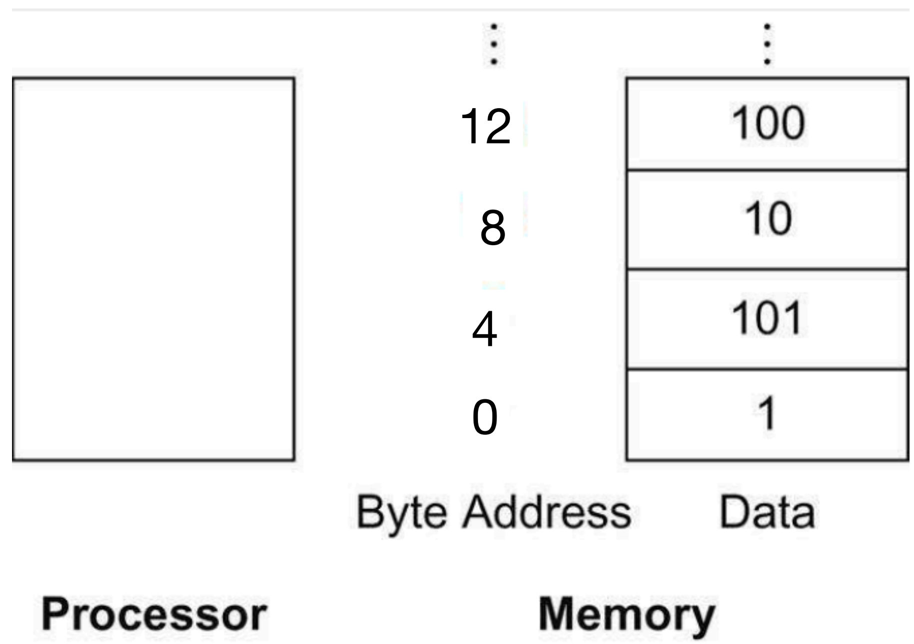
- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- Compiled RISC-V code:

```
lw      x9, 32(x22)
add     x9, x21, x9
sw      x9, 48(x22)
```



Registers vs. Memory

- Registers are faster to access than main memory and cash memory, and use less energy.
- Operating on memory data requires loads and stores.
- Compiler must use registers for variables as much as possible. Only spill to memory for less frequently used variables.
- Most programs have more variables than registers.

Immediate Operands

- Common to use a constant in operations
- Example: Add the constant 4 to register x22, assuming that x3 + AddrConstant4 is the memory address of the constant 4.

```
lw x9, AddrConstant4(x3)
add x22, x22, x9
```

- Alternative that avoids the load instruction - *add immediate*

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

- Design Principle 3: Make the common case fast
    - Small constants are common
    - Immediate operand avoids a load instruction

Representation of Numbers

The familiar way of representing numbers is to use 10 digits (0, 1, … , 9), such numbers are called base-10, or decimal numbers, e.g. 4831

The position of each digit represents a “power'” of the base (10):

$$4831 = 4 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$$

In logic circuits it is awkward to directly represent digits like 4, 8, and 3

We want only 2 digits: 0, 1 Therefore we will use base-2, or binary numbers

Unsigned Binary Integers

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ \ldots\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 64 bits: 0 to +18,446,774,073,709,551,615
- Using 32 bits: 0 to +4,294,967,295

Converting Decimal to Binary

Want to convert decimal number $D = d_{k-1}...d_1d_0$ with value $(V)10$, to binary number $B = b_{n-1}...b_2b_1b_0$

We would have:
$V = b_{n-1} \times 2^{n-1} + ... + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$

We next note that if we divide V by 2 we get:
$V/2 = b_{n-1} \times 2^{n-2} + ... + b_2 \times 2^1 + b_1 \times 2^0 + b_0 /2 = Q1 + b_0 /2$

With integer division, no fractions: just quotient and remainder.
Above, Q1 is the quotient and b_0 in {0,1} is the remainder (i.e. $b_0 = V - Q1 \times 2$)

Converting Decimal to Binary II

Thus, if we divide $(V)_{10}$ by 2, the remainder is b_0, the LSB of B

We next note Q_1 is also a binary number

If we divide Q_1 by 2, we get b_1 as the remainder

If we repeat until our quotient is 0, we can extract every digit of B

<u>857 Decimal To Binary Conversion:</u>

step 1 Perform the successive MOD operation by 2 for the given decimal number 857 and note down the remainder (either 0 or 1) for each operation. The last remainder is the MSB (most significant bit) and the first remainder is the LSB (least significant bit).

857 / 2 = 428 : Remainder is 1 → **LSB**
428 / 2 = 214 : Remainder is 0
214 / 2 = 107 : Remainder is 0
107 / 2 = 53 : Remainder is 1
53 / 2 = 26 : Remainder is 1
26 / 2 = 13 : Remainder is 0
13 / 2 = 6 : Remainder is 1
6 / 2 = 3 : Remainder is 0
3 / 2 = 1 : Remainder is 1
1 / 2 = 0 : Remainder is 1 → **MSB**

step 2 Write the remainders from MSB to LSB provide the equivalent binary number
1101011001

$857_{10} = 1101011001_2$

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

Octal and Hexadecimal Numbers

We have looked at radixes (bases) 10 and 2 so far

Can have any radix r

Thus have number
$K = (k_{n-1}\ k_{n-2} \ldots\ k_1\ k_0)_r$

For computers, we are interested in
octal (radix-8)
and
hexadecimal (radix-16)

Dec	Hex	Oct	Bin
0	0	000	0000
1	1	001	0001
2	2	002	0010
3	3	003	0011
4	4	004	0100
5	5	005	0101
6	6	006	0110
7	7	007	0111
8	8	010	1000
9	9	011	1001
10	A	012	1010
11	B	013	1011
12	C	014	1100
13	D	015	1101
14	E	016	1110
15	F	017	1111

Octal and Hexadecimal Numbers II

Why do we care about hexadecimal?

- Easy to convert between binary and hexadecimal

Hexadecimal is more compact thus easier for humans to use.

A 16 digit binary number is only a 4 digit hexadecimal number!

The C language uses the ***0x**nnnn* notation for hexadecimal numbers.

Signed Numbers

Binary number: $b_{n-1}b_{n-2} \ldots b_1 b_0$
b_{n-1} - sign
$b_{n-2} \ldots b_1 b_0$ - magnitude

For an n-bit signed number, the leftmost bit is used as the sign bit.

The number is negative when $b_{n-1} = 1$

The number is positive when $b_{n-1} = 0$

2s-Complement Signed Integers

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

Range: -2^{n-1} to $+2^{n-1} - 1$

Example (32 bit signed number):

$1111\ 1111\ 1111\ 1100_2 =$

$-1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0} =$

$-2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

Using 32 bits: -2,147,483,648 to 2,147,483,647

Using 64 bits: -9,223,372,036,854,775,808
to 9,223,372,036,854,775,807

2s-Complement Signed Integers

- Bit 31 is sign bit - 1 for negative numbers, 0 for non-negative numbers

- $-(-2^{n-1})$ can't be represented

- Some specific numbers
  - 0:   0000 0000 … 0000
  - - 1:   1111 1111 … 1111

  - Most-negative: 1000 0000 0000 0000
  - Most-positive: 0111 1111 1111 1111

Signed Negation

- Complement and add 1

Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

Example: negate +2

0000 0000 … 0010
1111 1111 … 1101 (complement)
1 (add 1)
— — — — — — — — —
1111 1111 … 1110 = -2_{10}

Sign Extension

- Representing a number using more bits we need to preserve the numeric value.
- Replicate the sign bit to the left c.f. unsigned values: extend with 0s.

- Examples: 8-bit to 16-bit

+2: 0000 0010 => 0000 0000 0000 0010
–2: 1111 1110 => 1111 1111 1111 1110

- In RISC-V instruction set

`lb:` sign-extend loaded byte
`lbu:` zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary called machine code.

- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …

RISC-V Instruction Types

R-type (R for registers) instructions use 3 register operands (2 source registers and one destination register).

I-type (I for immediate) instructions use 2 register operands (1 source, 1 destination) and one 12-bit immediate.

S-type (S for stores) instructions use 2 register operands (2 source) and one 12-bit immediate.

SB-type (conditional Branch, fields like the S)

U-type (Upper immediate format)

UJ-type (Unconditional jump)

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)

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

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

$015A04B3_{16}$

RISC-V Simulator: https://www.kvakil.me/venus/

RISC-V I-format Instructions

immediate[11:0]	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
- Design Principle 3: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-format Instructions

immediate[11:5]	rs2	rs1	funct3	immediate[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

RISC-V SB-format Instructions

Conditional Branch - Fields like S

immediate[12,10:5]	rs2	rs1	funct3	immediate[4:1,11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

if (R[rs1]!=R[rs2])
    PC = PC+{imm,1b'0'}

`bne x10, x11, 2000` // if x10 != x11 , go to location 2000

{imm, 1b'0} denotes concatenating immediate, imm, with one bit of value 0)

RISC-V U-format Instructions

Upper immediate format

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

R[rd] = PC+{imm,12b'0'}

`auipc` - (used for position independent code, such as for dynamic linked libraries, DLL)

RISC-V UJ-format Instructions

Unconditional jump, fields like the U

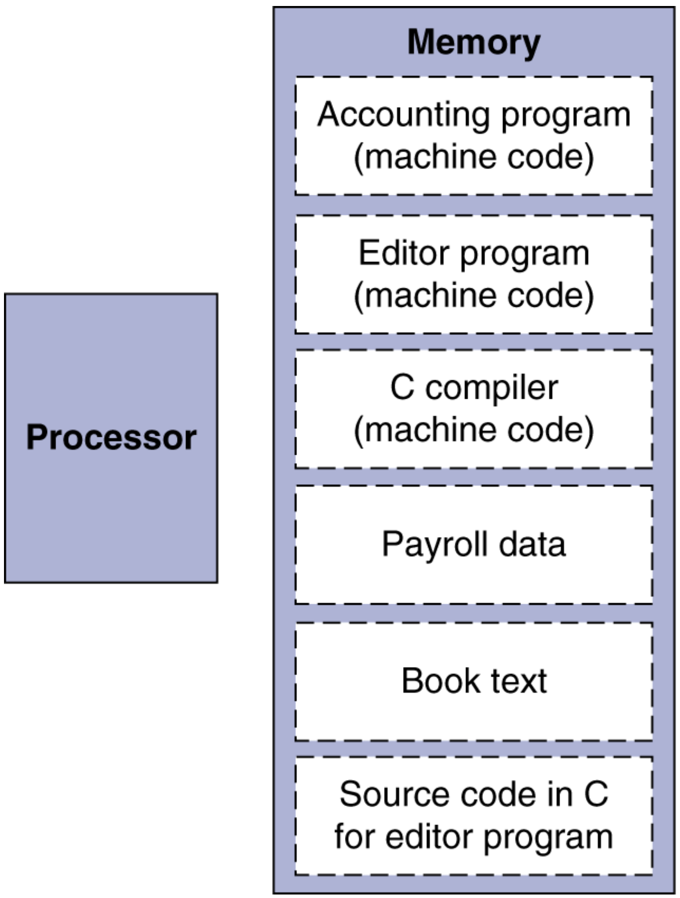
immediate[20,10:1,11,19:12]	rd	opcode
20 bits	5 bits	7 bits

R[rd] = PC+4; PC + {imm,1b'0'}

instruction `jal`, used for jump and link instruction (where 'link' means it will return to where it was called, by placing address or link into register x11 which holds the return address)- typically used for procedure calls

`jal x11, 1000` // go to location 1000

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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	~	~	`xor, xori`

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)

AND Operations

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

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

x10	00000000 00000000 00001101 11000000
x11	00000000 00000000 00111100 00000000
x9	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 00001101 11000000
x11	00000000 00000000 00111100 00000000
x9	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	00001101	11000000
x12	11111111	11111111	11111111	11111111
x9	11111111	11111111	11110010	00111111

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

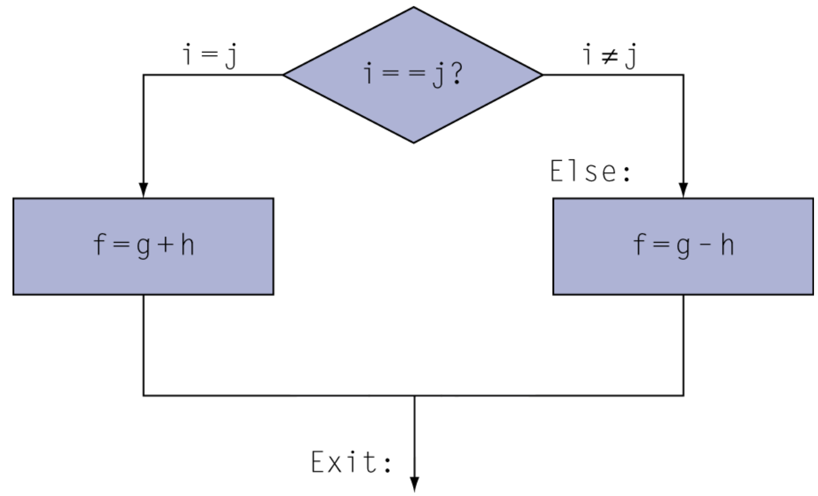
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g,h,i,j… in x19, x20,…x23



- Compiled RISC-V code:

```
      bne x22, x23, Else
      add x19, x20, x21 // f = g + h
      beq x0, x0, Exit // unconditional
Else: sub x19, x20, x21 // f = g -h
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

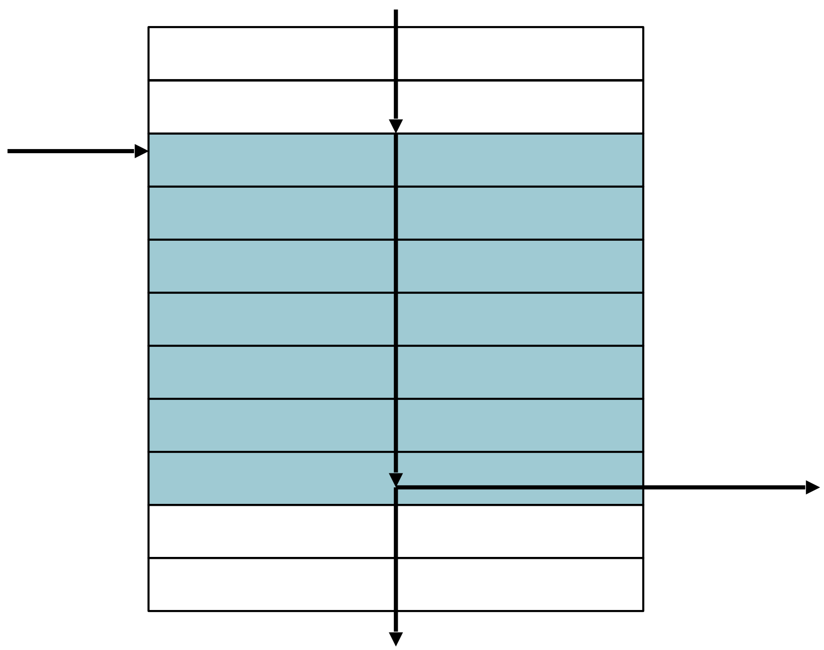
  - *i* in x22, *k* in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 2    // x10 = i * 4
      add  x10, x10, x25  // x10 = address of save[i]
      lw   x9, 0(x10)     // x9 = save[i]
      bne  x9, x24, Exit  // go to Exit if save[i]!=k
      addi x22, x22, 1    // i = i+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`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - 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:
```

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

Procedure Calling

A procedure is a stored subroutine that performs a specific task based on the parameters passed to it

  - It may also provide a return value

- The program that executes the procedure is called the caller

The procedure being executed is called the callee

Steps required to execute procedure

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
6. Return to place of call (address in x1)

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
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps e.g., for case/switch statements

Leaf Procedure Example

- C code:

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

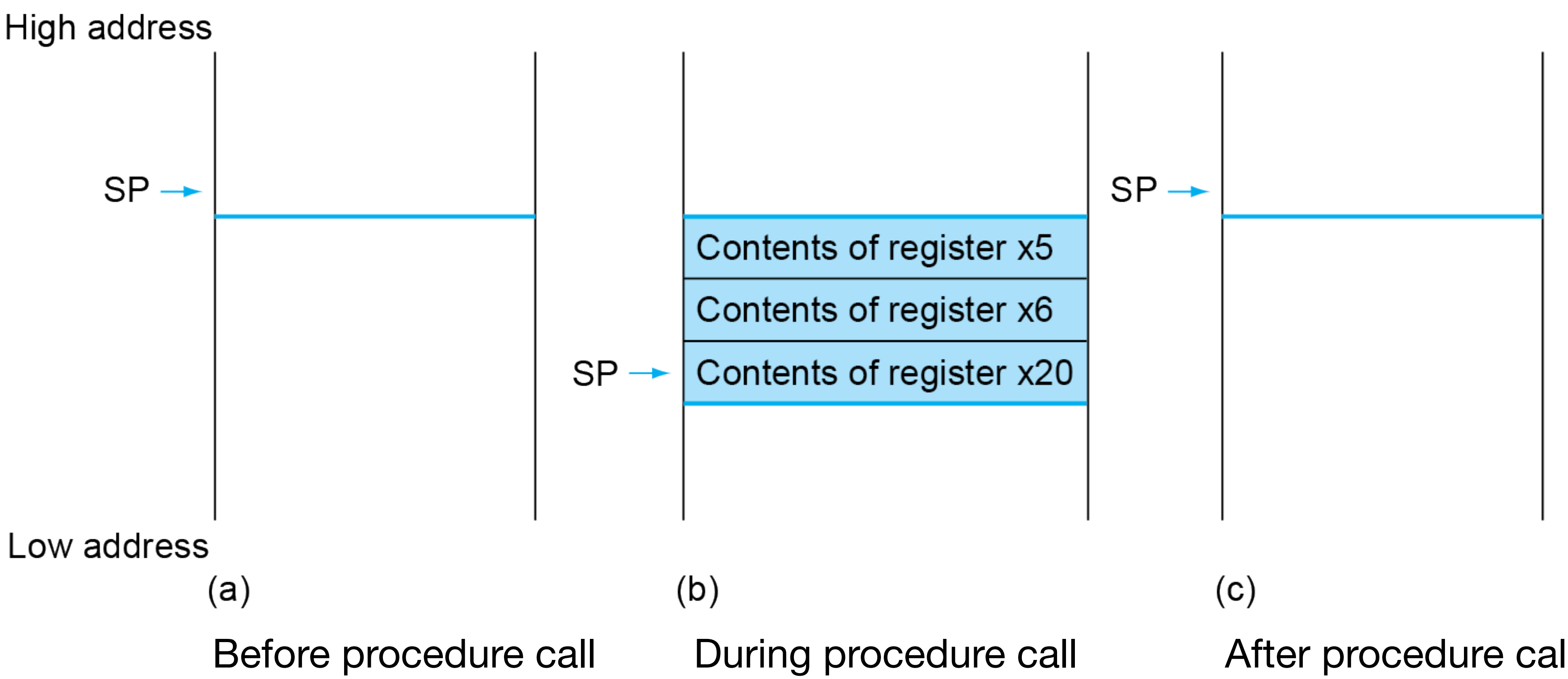
- Arguments *g*, *h*, *i*, *j* in x10, …, x13, *f* in x20
- Temporaries x5, x6
- Need to save x5, x6, x20 on stack

Leaf Procedure Example

- RISC-V code 32bit address:

```
leaf_example:                // label of the procedure
   addi sp,sp,-12            // adjust stack to make room for 3 items
   sw   x5,8(sp)             // save register x5
   sw   x6,4(sp)             // save register x6
   sw   x20,0(sp)            // save register x20
   add  x5,x10,x11           // register x5 contains g + h
   add  x6,x12,x13           // register x6 contains i + j
   sub  x20,x5,x6            // f = x5 - x6
   addi x10,x20,0            // returns f (x10 = x20 + 0)
   lw   x20,0(sp)            // restore register x20 for caller
   lw   x6,4(sp)             // restore register x6 for caller
   lw   x5,8(sp)             // restore register x5 for caller
   addi sp,sp,12             // adjust stack to delete 3 items
   jalr x0,0(x1)             // branch back to calling routine
```

Local Data on the Stack



The values of the stack pointer and the stack

Register Usage

In last example, we stored to stack all registers that we used

Don't want to store registers that don’t contain needed data

- 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
- In example above x5 and x6 are saved on stack. Caller should preserve those values before procedure is called.

Register Usage II

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

- Above shows which registers must be saved, and which do not

Q: Parameters are passed in 8 registers x10-x17. What if there are more than 8 parameters, how they are passed?

Non-Leaf (Nested) Procedures

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

- *See "Nested Procedures" section of text for more details*

Non-Leaf Procedure Example

- C code:

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

  - Argument *n* in x10
  - Result in x10

https://en.wikipedia.org/wiki/Factorial

Non-Leaf Procedure Example

- RISC-V code:

```
fact:
    addi sp,sp,-8       // adjust stack for 2 items
    sw   x1,4(sp)        // save return address
    sw   x10,0(sp)       // save the argument n
    addi x5,x10,-1       // x5 = n -1
    bge  x5,x0,L1        // if (n - 1) >= 0, go to L1
    addi x10,x0,1        // return 1
    addi sp,sp,8         // pop 2 items from stack
    jalr x0,0(x1)        // return to caller
L1: addi x10,x10,-1      // n>=1: argument gets n-1
    jal  x1,fact          // call fact with n-1
    addi x6,x10,0         // return from jal,move result of fact(n-1) to x6
    lw   x10,0(sp)        // restore argument n
    lw   x1,4(sp)         // restore the return address
    addi sp,sp,8          // adjust stack pointer to pop 2 items
    mul  x10,x10,x6       // return n * fact (n - 1)
    jalr x0,0(x1)         // return to the caller
```

Local Variables on Stack

Variables that are local to a procedure are called automatic variables

- They are created when procedure starts and destroyed when procedure exits

What if not enough registers free, or variable type doesn't fit in a register (i.e. an array or structure)?

- Then the local variable is created on the stack

Local Variables on Stack II

Segment of stack containing procedures saved registers and local variables is called a procedure frame or activation record

Some compilers use a frame pointer (register FP or X8)

- Points to the first word of the procedure's frame
- Provides a stable base register for local variable access

Can just use SP, but makes variable access more complicated

Stack with FP

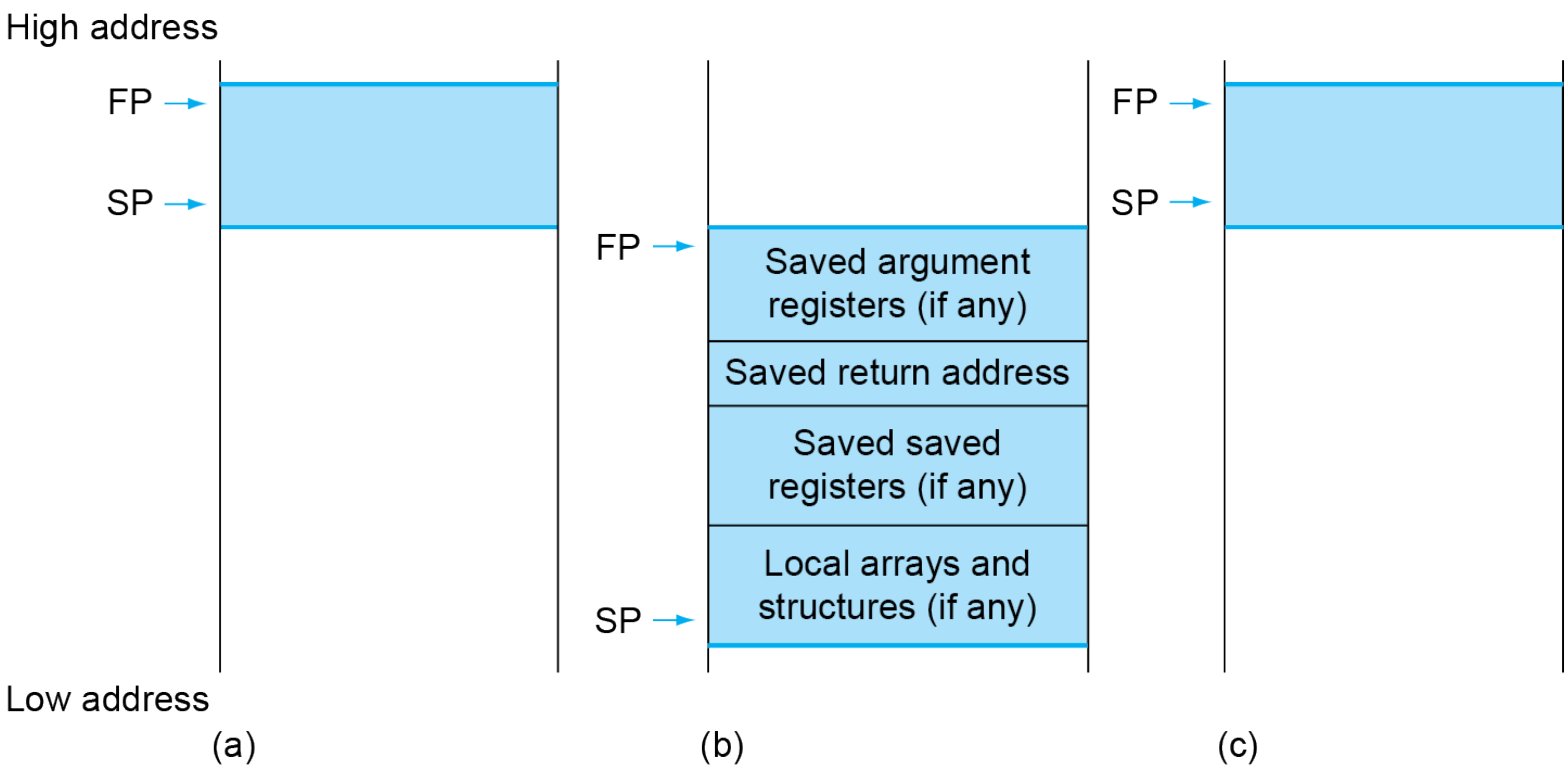


Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.

Variables and the Heap

In addition to automatic variables, we need to allocate static data (i.e. constants) and dynamic data structures

Dynamic data structures are used for variables whose size can change over time (i.e. a linked list)

Dynamic data is stored in a section of the memory called the heap

The C language allocates space on the heap with the `malloc()` function, and frees it with the `free()` function.

Memory Layout

The next slide shows memory convention for allocating memory for use with Linux operating system

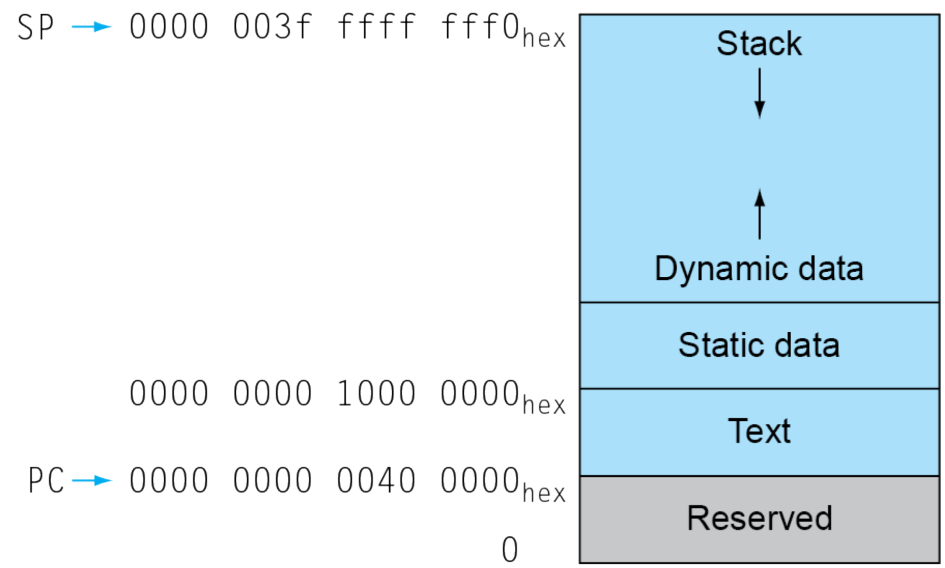
The stack starts at the high end of memory space and grows down

The heap starts at low end of memory and grows to meet the stack

There is also an area for static data, machine code (called text segment), and a reserved area

Memory Layout II

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



Q1:What is "memory leak"?
Q2:What are "dangling pointers"?

RISC-V registers

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Character Data (8 bits)

- Common for computers to store text data
- Most use 8 bits to represent characters
- Most common is the American Standard code for Information Interchange (ASCII)
- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
      https://en.wikipedia.org/wiki/ISO/IEC_8859-1

ASCII Representation of Characters

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

- ASCII only uses the rightmost 7 bits, the eighth is unspecified
- Not shown values are control characters such as tab and backspace

Character Data (32 bits)

- The characters of some human languages do not fit 8 bits
  - Need larger format

- 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

Byte Operations

- RISC-V byte load/store
  - Load byte: Sign extend to 32 bits in rd
    `lb rd, offset(rs1)`
    - Stores 1 byte in rightmost 8 bits of register
  - Load byte unsigned: Zero extend to 32 bits in rd
    `lbu rd, offset(rs1)`
  - Store byte: Store rightmost 8 bits
    `sb rs2, offset(rs1)`
    - Store just rightmost byte into memory

Halfword Operations

- RISC-V halfword load/store
  - Load halfword: Sign extend to 32 bits in rd

    ```
    lh rd, offset(rs1)
    ```

    - Stores 2 bytes in rightmost 16 bits of register

  - Load halfword unsigned: Zero extend to 32 bits in rd

    ```
    lhu rd, offset(rs1)
    ```

  - Store halfword: Store rightmost 16 bits

    ```
    sh rs2, offset(rs1)
    ```

    - Store two rightmost byte into memory

Word Operations

- RISC-V word load/store

  - Load word: Sign extend to 32 bits in rd

    ```
    lw rd, offset(rs1)
    ```

    - Stores 4 byte in rightmost 32 bits of register

  - Load word unsigned: Zero extend to 32 bits in rd

    ```
    lwu rd, offset(rs1)
    ```

  - Store word: Store rightmost 32 bits

    ```
    sw rs2, offset(rs1)
    ```

    - Store four rightmost byte into memory

String Copy Example

- Characters are normally grouped into strings, which have a variable length, i.e. “Hello”
- The C language uses the null character ‘\0' (zero) to mark the end of the string - null-terminated string
- C code:

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

String Copy Example

- RISC-V code (base addresses for arrays `x` and `y` are found in `x10` and `x11` while `i` is in `x19`.

```
strcpy:
 addi sp,sp,-4          //adjust stack for 1 w
 sw   x19,0(sp)        // save x19 on stack
 add  x19,x0,x0         // sets x19 to 0, 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             // next iteration of loop
L2: lw   x19,0(sp)      // restore saved x19
 addi sp,sp,4           // pop 1 w from stack
 jalr x0,0(x1)          // and return
```

32-bit Constants Example

- Most constants are small, 12-bit immediate is sufficient, but we may have bigger constants

Q: How to load 32 bit constant to X19?

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

A: First, we would load bits 12 through 31 with that bit pattern, which is 976 in decimal, using `lui`:

`lui x19, 976` // 0x003D0

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

The next step is to add in the lowest 12 bits 0x500, whose decimal value is 1280:

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

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

Branch Addressing

The RISC-V branch instructions use the RISC-V instruction format called SB-type.

This format can represent branch addresses from –4096 to 4094, only possible to branch to even addresses

SB-type format

Example: The instruction “if x10 != x11, go to location 0111 1101 0000”

```
bne x10, x11, 2000
```

could be assembled into format

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

Jump Addressing

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

- UJ-type format:

Example: The instruction

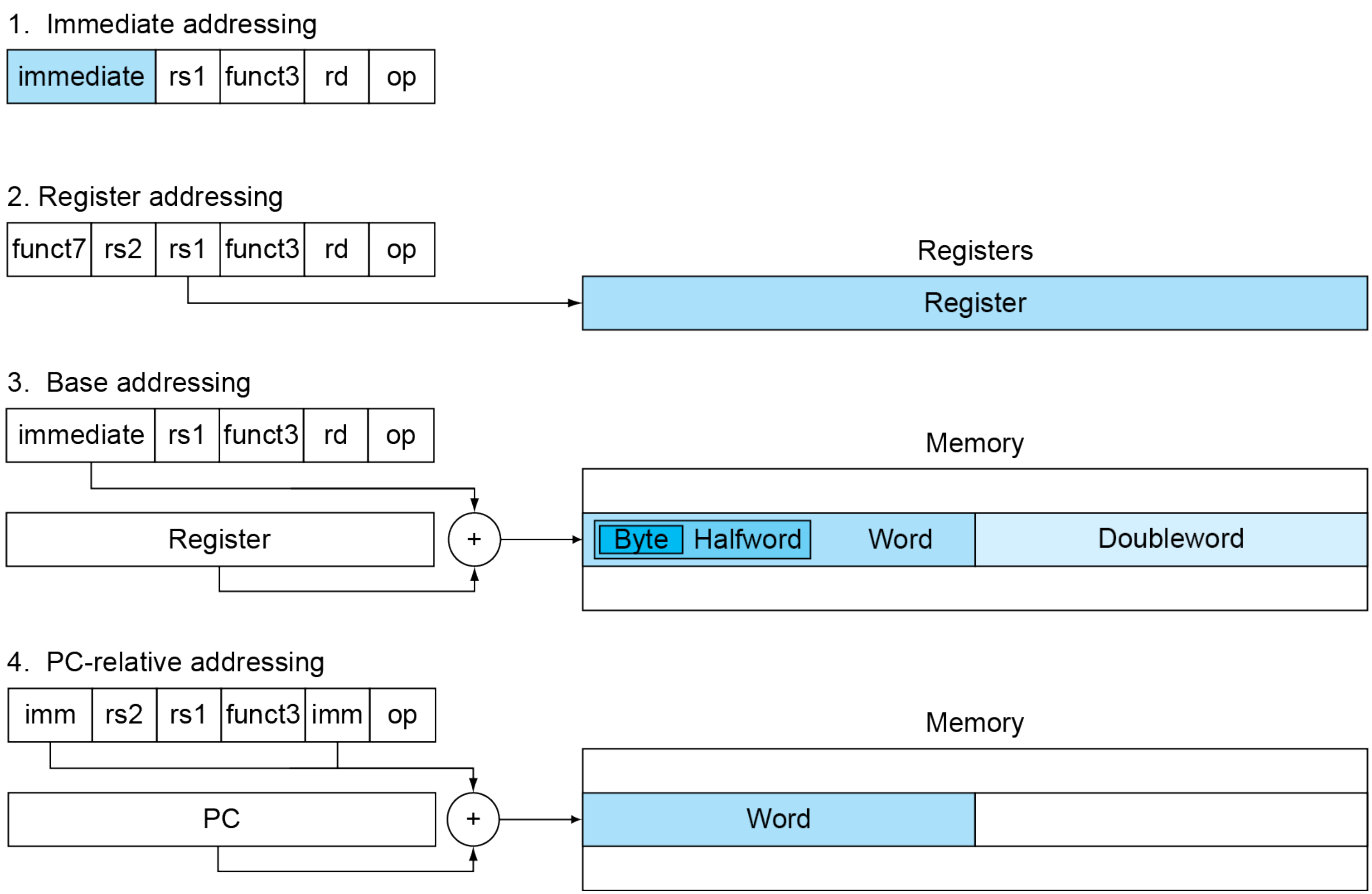
`jal x0, 2000 // go to 2000`$_{ten}$ `= 0111 1101 0000`

is assembled into format

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

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

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

R type instructions (R for registers): R[rd] = R[rs1] + R[rs2]
I type instructions (I for immediate): R[rd] = M[R[rs1] + imm](63:0)
S type instructions (S for stores): M[R[rs1] + imm](63:0) = R[rs2](63:0)
SB type instructions (conditional branch, fields like the S):
if(R[rs1] != R[rs2]) PC = PC + { imm,1b'0}
U type instructions (upper immediate format): R[rd] = PC + { imm,12'b0}
UJ type instructions (unconditional jump, fields like the U):
R[rd] = PC + 4; PC = PC + { imm,1b'0}

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

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 on the memory location specified by the contents of x20 (to test/set lock variable)

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

Example 2:

Acquire a lock at the location in register x20

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

To release the lock (unlock) use a regular store to write 0 into location

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

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

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

x10 - array base address

x11 - size

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

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

x10 - array base address

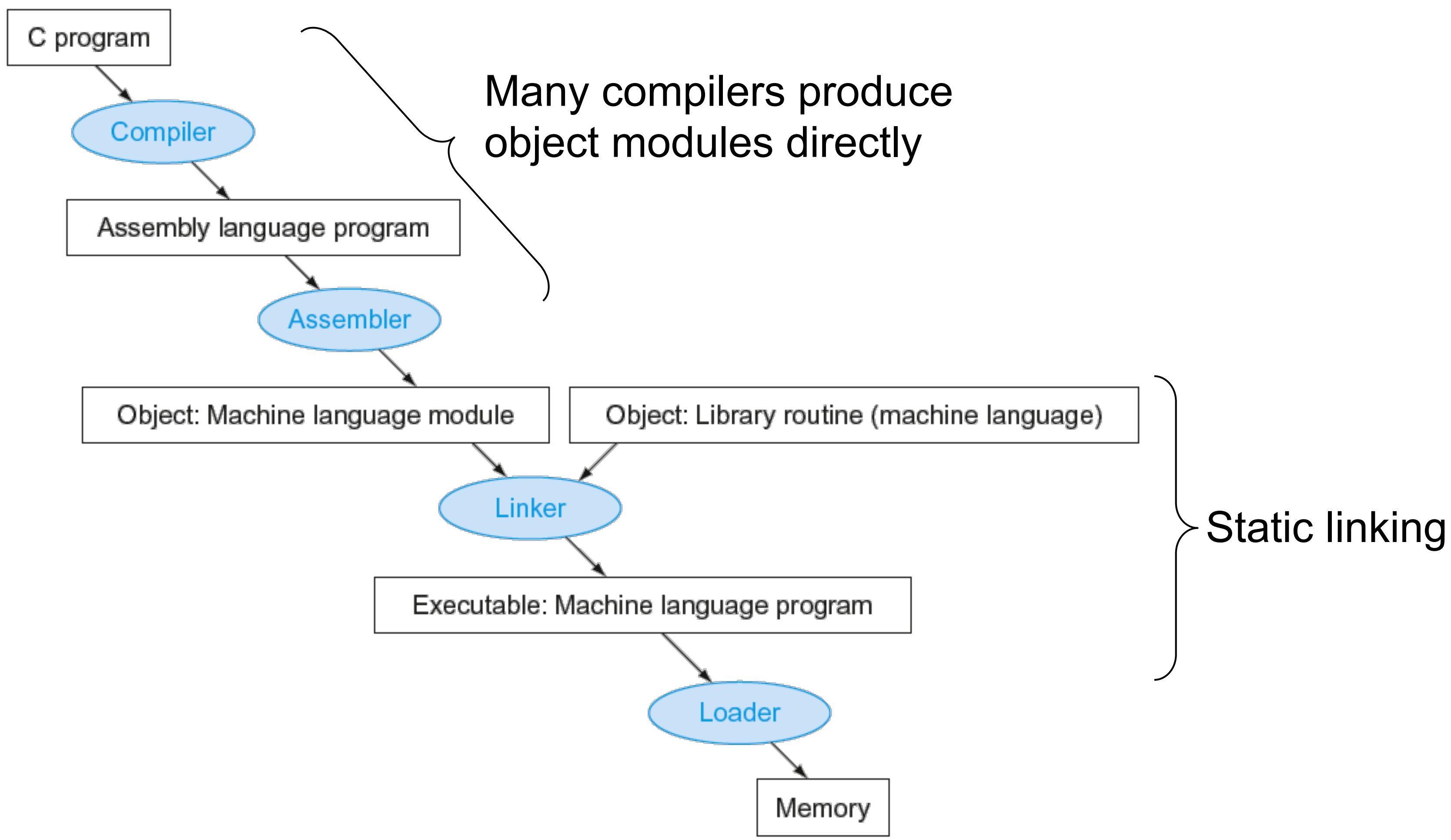
x11 - size

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

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory, or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including sp, fp, gp)
  6. Jump to startup routine
     - Copies arguments to x10, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
- Requires procedure code to be relocatable
- Avoids image bloat caused by static linking of all (transitively) referenced libraries
- Automatically picks up new library versions

Other RISC-V Instructions

- Base integer instructions
  - Those previously described, plus
  - `auipc rd, immed` // rd = (imm<<12) + pc
    - follow by `jalr` (adds 12-bit immed) for long jump
  - `slt, sltu, slti, sltiu`: set less than (like MIPS)

- RV64I (only)
  - `addw, subw, addiw`: 32-bit add/sub
  - `sllw, srlw, slliw, srliw, sraiw`: 32-bit shift

Instruction Set Extensions

- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
    - 16-bit encoding for frequently used instructions

Concluding Remarks

Design principles

1. Simplicity favours regularity
2. Smaller is faster
3. Good design demands good compromises

Make the common case fast

Layers of software/hardware

- Compiler, assembler, hardware

RISC-V: typical of RISC ISAs

Additional Reading

Read for your own interest

2.16 Real Stuff: MIPS Instructions

2.17 Real Stuff: x86 Instructions

2.19 Fallacies and Pitfalls

2.20 Concluding Remarks

Thank You