Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

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
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c   // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled RISC-V code:

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
add f, t0, t1  // f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 32-bit register file
  - Use for frequently accessed data
  - 32-bit data is called a “word”
    - 32 x 32-bit general purpose registers x0 to x31
- Another version has 32 × 64-bit register file
  - 64-bit data is called a “doubleword”

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

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);
```

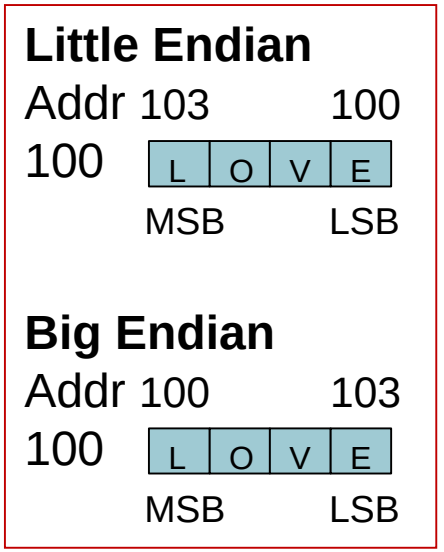
  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

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

Memory Operands

- 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
- RISC-V is Little Endian
  - Least-significant byte at the least address of a **word**
  - *c.f.* Big Endian: most-significant byte at the least address
- RISC-V does not require words to be aligned in memory (i.e., a word's address may not be a multiple of 4)
  - Unlike some other ISAs



Memory Operand Example

- C code:

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

  - h in x21, **base address** of A in x22

- Compiled RISC-V code:
  - Index 8 requires an offset of 32
    - 4 bytes per word

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

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 (**spill**: move data in a register to memory)
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

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

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Unsigned Binary Integers

- Given an n-bit number

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

- 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 32 bits: 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits: −2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers cont.

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented with *n* bits
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

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

1s' complement plus one is equal to 2's complement.

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

Sign Extension

- Representing a number using more bits
  - 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, …
  - Regularity!

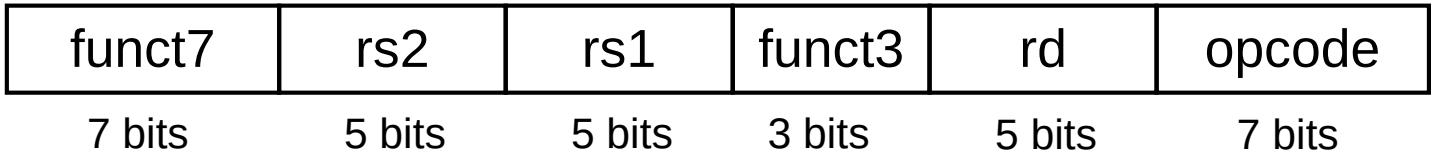
Hexadecimal

- 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

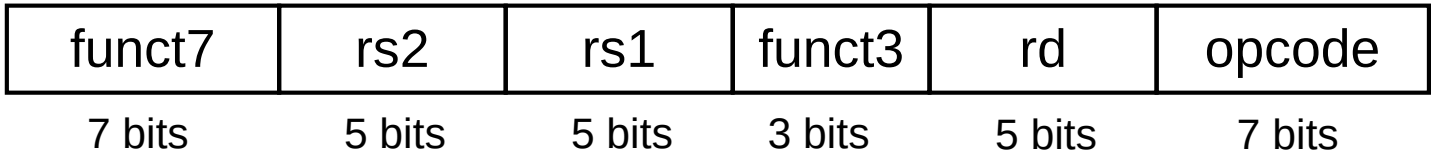
- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions



- 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



rd rs1 rs2

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

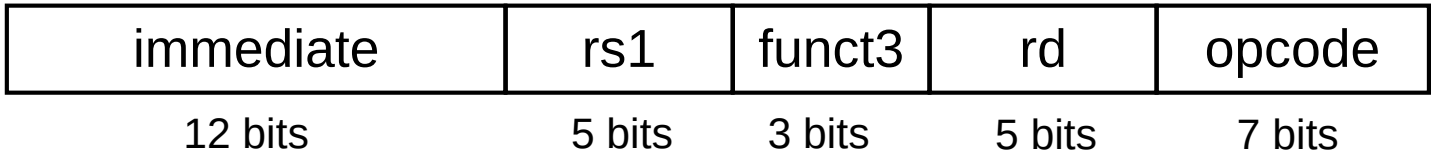
0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

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

$015A04B3_{16}$

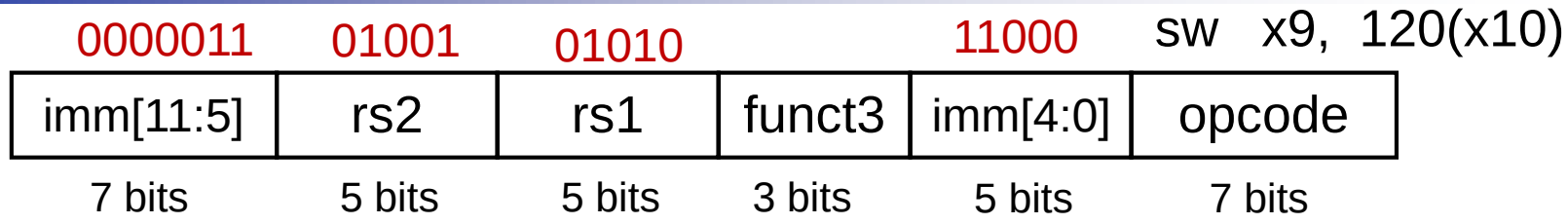
RISC-V I-format Instructions



- 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: lw x9, 32(x22)

  rd **imm** rs1
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow all instructions having a uniform length of 32 bits
  - Keep formats as similar as possible

RISC-V S-format Instructions

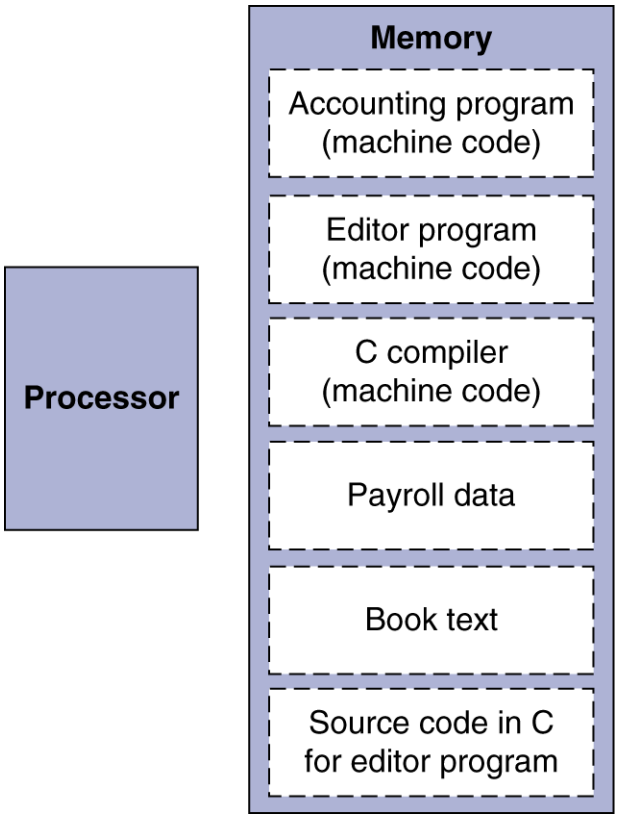


- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place
- Translating A[30] = h + A[30] + 1; Assuming **h** is in x21 and the base of **A** is in x10.
  - lw x9, 120(x10) //120(x10) is the address of A[30]
  - add x9, x21, x9
  - addi x9, x9, 1
  - sw x9, 120(x10)

    rs2 rs1

Stored Program Computers

The BIG Picture



- 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

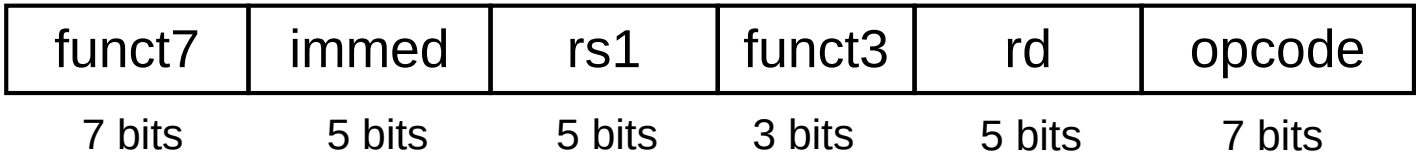
Logical Operations

- 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	~	~	

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

Shift Operations

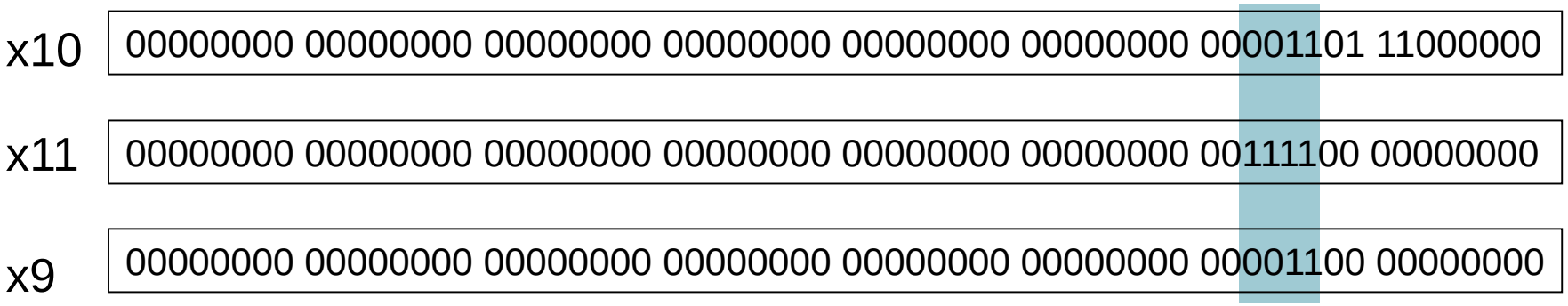


- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

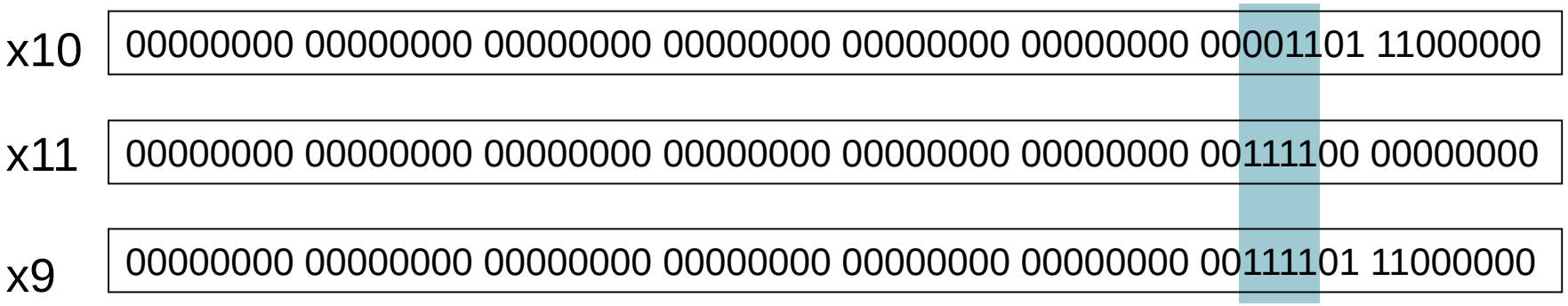
```
and x9,x10,x11
```



OR Operations

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

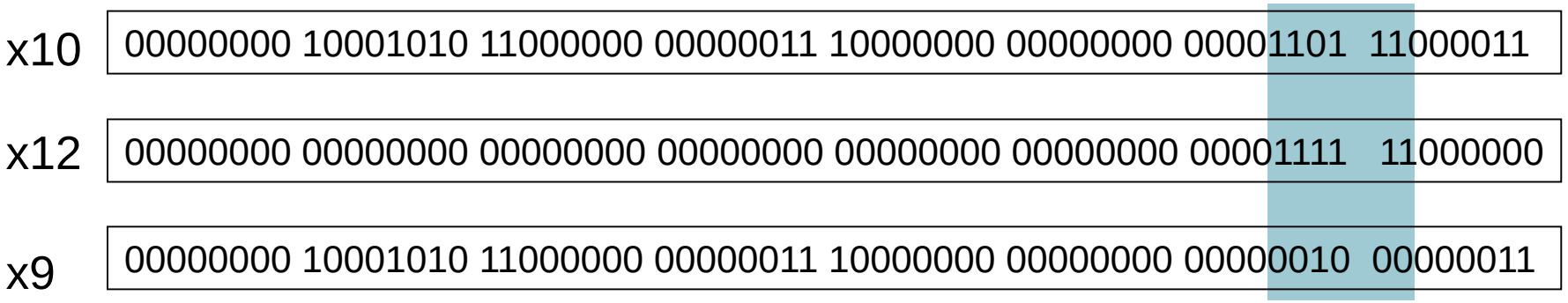
```
or x9,x10,x11
```



XOR Operations

- Differencing operation
  - Negate some bits, leave others unchanged

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



Conditional Operations

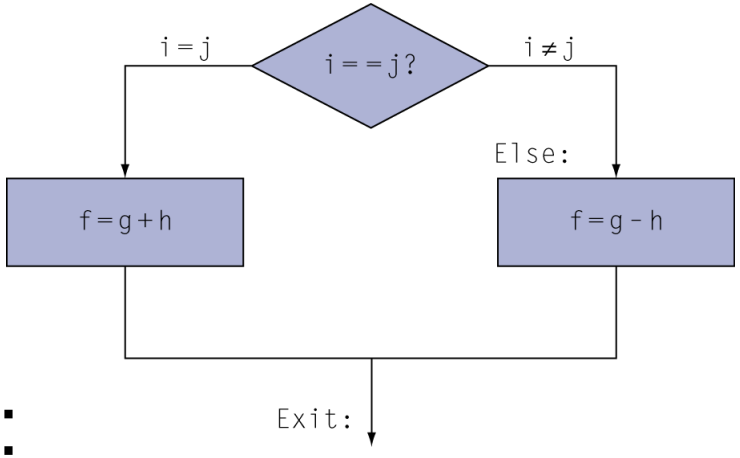
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

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

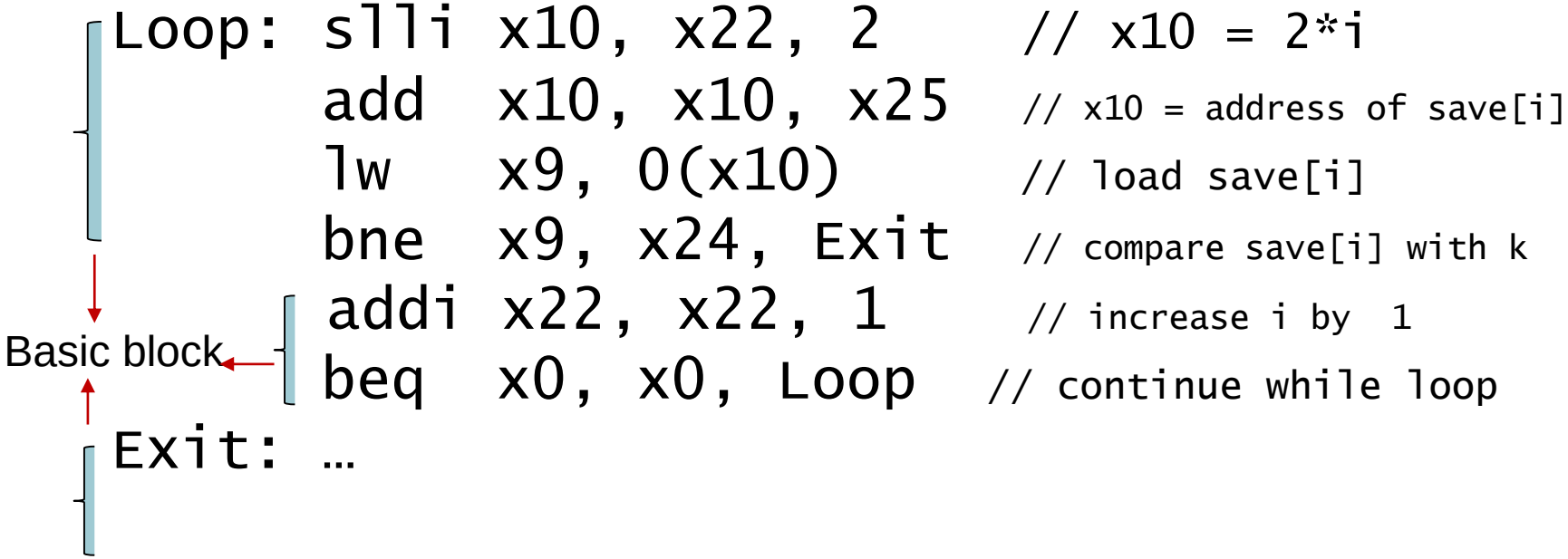
Assembler calculates addresses

Compiling Loop Statements

- C code:

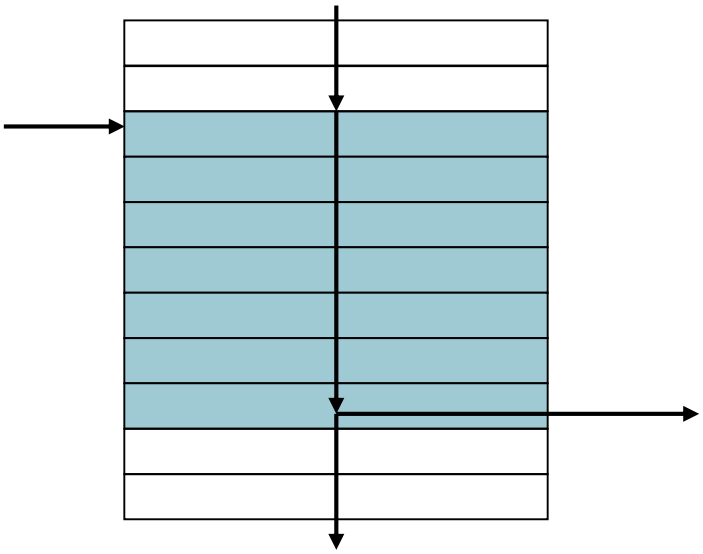
```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of **save** in x25
- Compiled RISC-V code:



Basic Blocks

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



- The number of times that each of the instructions in a basic block is executed are the same.
- A compiler identifies basic blocks for optimization.
- An advanced processor can accelerate execution of basic blocks by executing some of the instructions in the same basic in parallel.
- *All instructions in the same basic block will be executed in the same number of times.*

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 (function) Calling

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

X10 to X17 are used to pass arguments into a function.

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 with non-zero offset.

Leaf Procedure Example

- C code:

```
int leaf_example (int g,int h,int i,int j)
{
  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

- RISC-V code:

```
leaf_example:
   addi sp,sp,-12
   sw   x5,8(sp)
   sw   x6,4(sp)
   sw   x20,0(sp)
   add  x5,x10,x11
   add  x6,x12,x13
   sub  x20,x5,x6
   addi x10,x20,0
   lw   x20,0(sp)
   lw   x6,4(sp)
   lw   x5,8(sp)
   addi sp,sp,12
   jalr x0,0(x1)
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

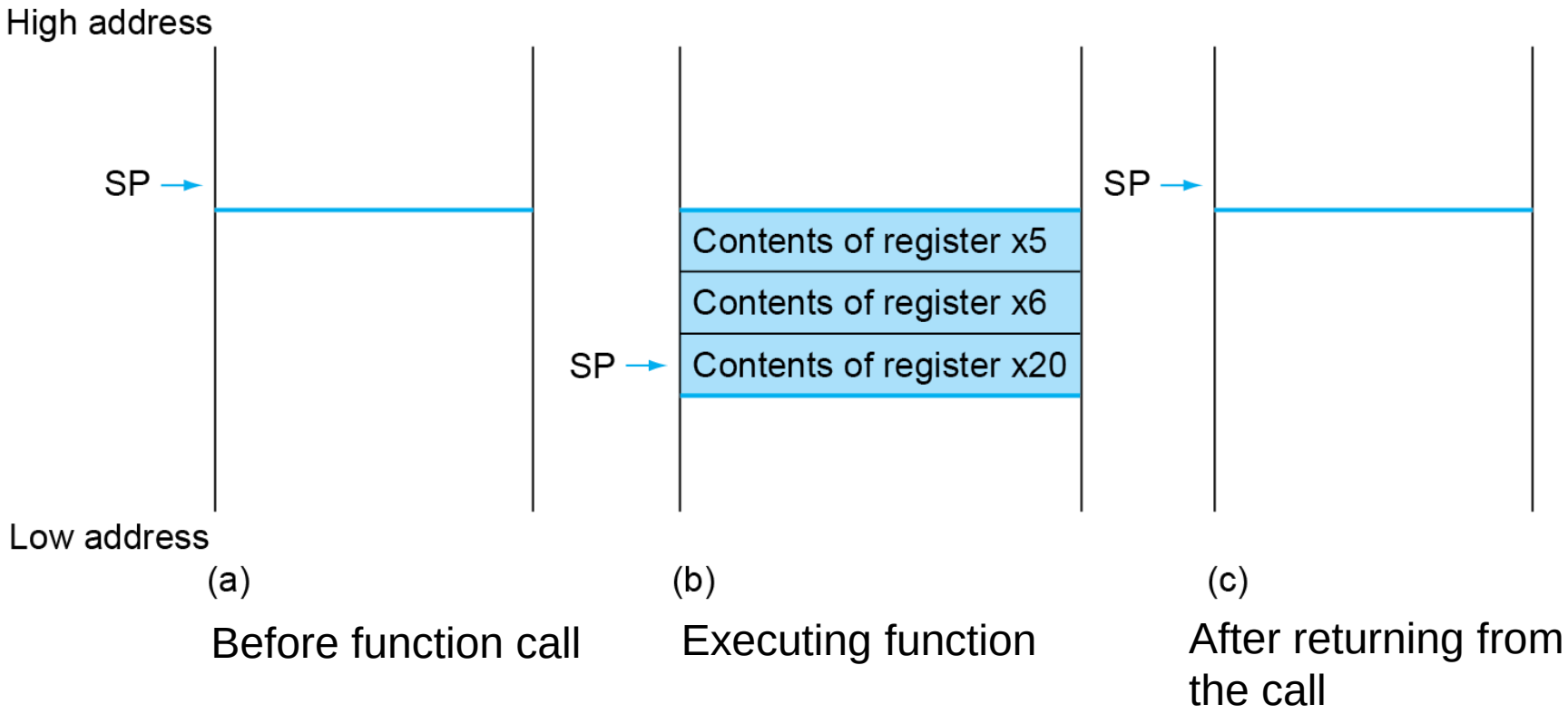
f = x5 – x6

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 (i.e., procedure being called)
  - In the above example, saving and restoring of x5 and x6 are not needed if this convention is followed.
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Non-Leaf Procedures

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

Non-Leaf Procedure Example

- C code: Fact(n) = n* fact(n-1)

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

  - Argument n in x10
  - Result in x10

Non-Leaf Procedure Example (1)

- RISC-V code:

```
fact:
    addi sp,sp,-8
    sw   x1,4(sp)
    sw   x10,0(sp)
    addi x5,x10,-1
    bge  x5,x0,L1
    addi x10,x0,1
    addi sp,sp,8
    jalr x0,0(x1)
L1: addi x10,x10,-1
    jal  x1,fact
RA: addi x6,x10,0
    lw   x10,0(sp)
    lw   x1,4(sp)
    addi sp,sp,8
    mul  x10,x10,x6
    jalr x0,0(x1)
```

Save return address to the caller and n on stack

x5 = n - 1

if n >= 1, go to L1

These two instructions check whether n >= 1

Else, set return value to 1

Pop stack, *don't bother restoring x1's values* because x1 still keeps RA

Return

n = n - 1

call fact(n-1)

move result of fact(n - 1) to x6; RA is return address of fact(n-1)

Restore caller's n

Restore caller's return address

Pop stack

return n * fact(n-1)

return

Non-Leaf Procedure Example (2)

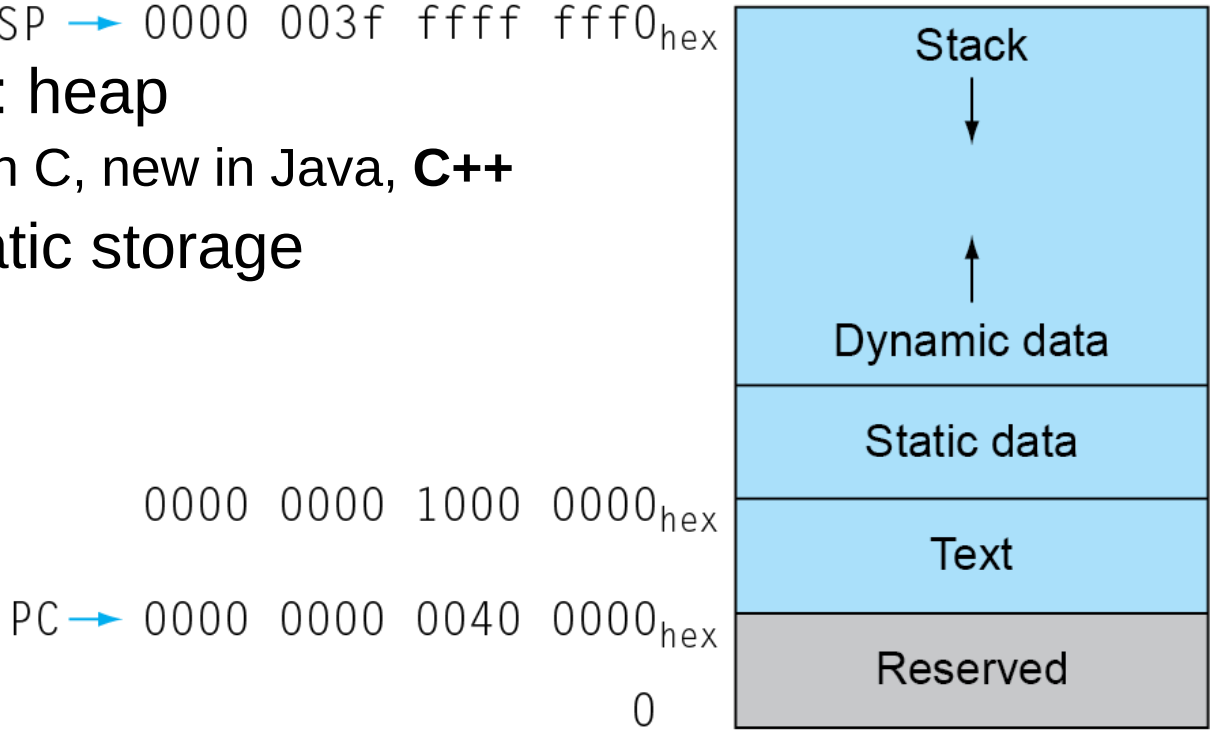
Assume n=4 and the caller make a call with return address CA

```
   fact:
1     addi sp,sp,-8
2     sw   x1,4(sp)
3     sw   x10,0(sp)
4     addi x5,x10,-1
5     bge  x5,x0,L1
6     addi x10,x0,1
7     addi sp,sp,8
8     jalr x0,0(x1)
9  L1: addi x10,x10,-1
10    jal  x1,fact
11 RA: addi x6,x10,0
12    lw   x10,0(sp)
13    lw   x1,4(sp)
14    addi sp,sp,8
15     mul  x10,x10,x6
16    jalr x0,0(x1)
```

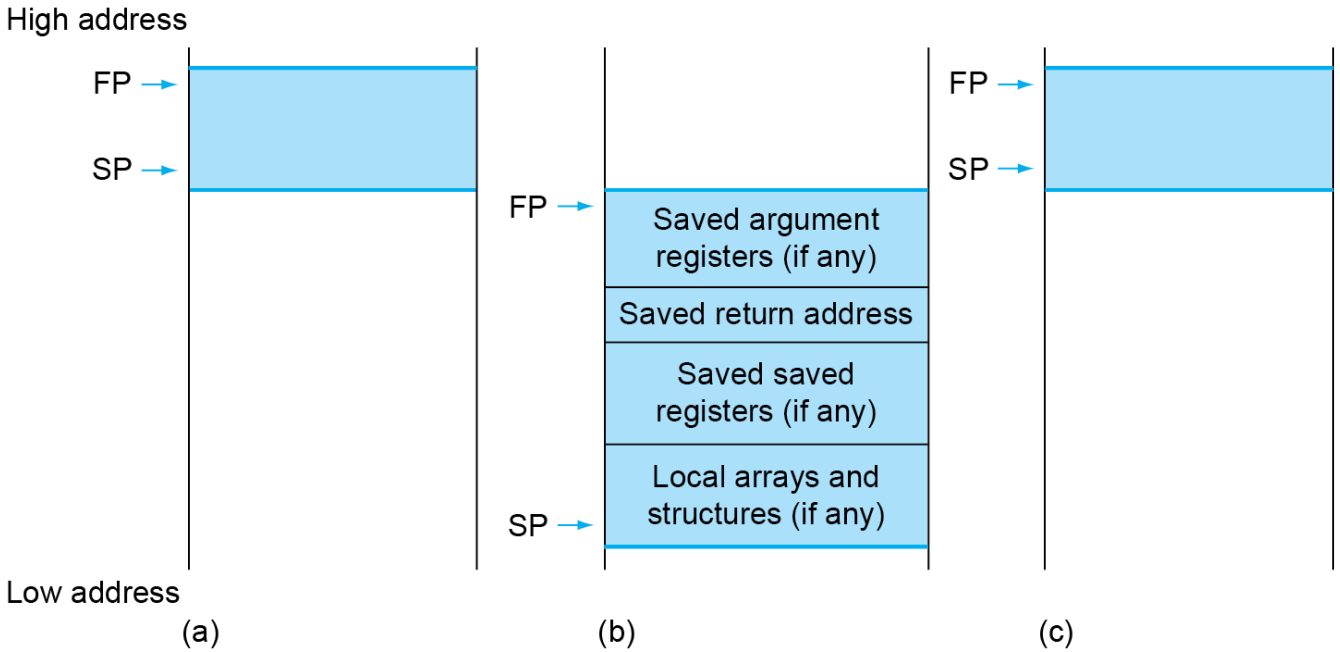
when called with n=4, inst 1~5, 9, 10 executed
x1=CA, x10=4 pushed into stack
when called with n=3, inst 1~5, 9, 10 executed
x1=RA, x10=3 pushed into stack
when called with n=2, inst 1~5, 9, 10 executed
x1=RA, x10=2 pushed into stack
when called with n=1, inst 1~5, 9, 10 executed
x1=RA, x10=1 pushed into stack
when called with n=0, inst 1~8 executed, set **x10 =1**, jump to RA (via inst 8), then
execute inst 11~16 for "return with n=1", set x6=1, x10=1, x1=RA, **x10=1** (doing x10 mul x6), then
execute inst 11~16 for "return with n=2", set x6=1, x10=2, x1=RA, **x10=2** (doing x10 mul x6), then
execute inst 11~16 for "return with n=3", set x6=2, x10=3, x1=RA, **x10=6** (doing x10 mul x6), then
execute inst 11~16 for "return with n=4", set x6=6, x10=4, x1=CA, **x10=24** (doing x10 mul x6), then
Retrun to the first caller with return address CA

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, **C++**
- 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

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

Byte/Halfword/Word Operations

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

Why does it not need to consider signed or unsigned data for *store* instructions?

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;
}
```

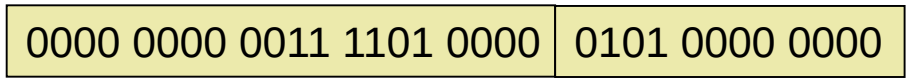
String Copy Example

- RISC-V code:

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

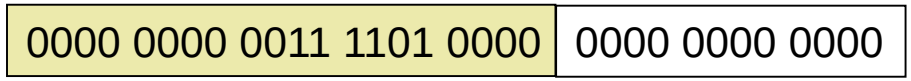
32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For an occasional 32-bit constant
  - First, using **lui rd, constant** to copy 20-bit constant to bits [31:12] of rd
  - Then, adds the lowest 12 bits
- For example, forming the constant

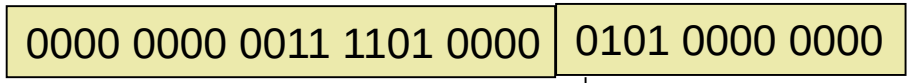


More convenient by pseudo instruction **li a0, CONSTANT**

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




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

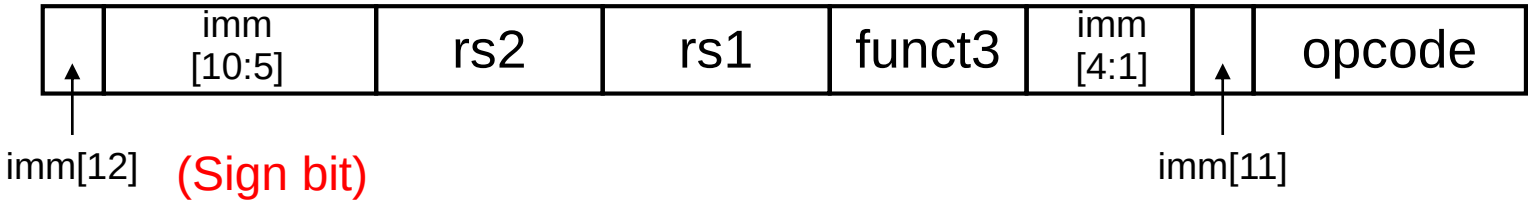


Sign bit is zero.

Read also the Elaboration in page 121 for the case with a 1 on the sign bit of an imm.

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format (for PC-relative Addressing)



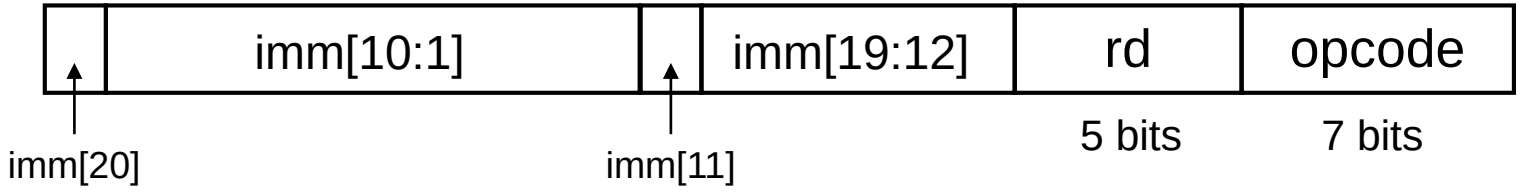
If (compare(rs1, rs2) is true)

Jump to **Target address** = PC + immediate × 2

i.e., -4096 to 4094

Jump Addressing

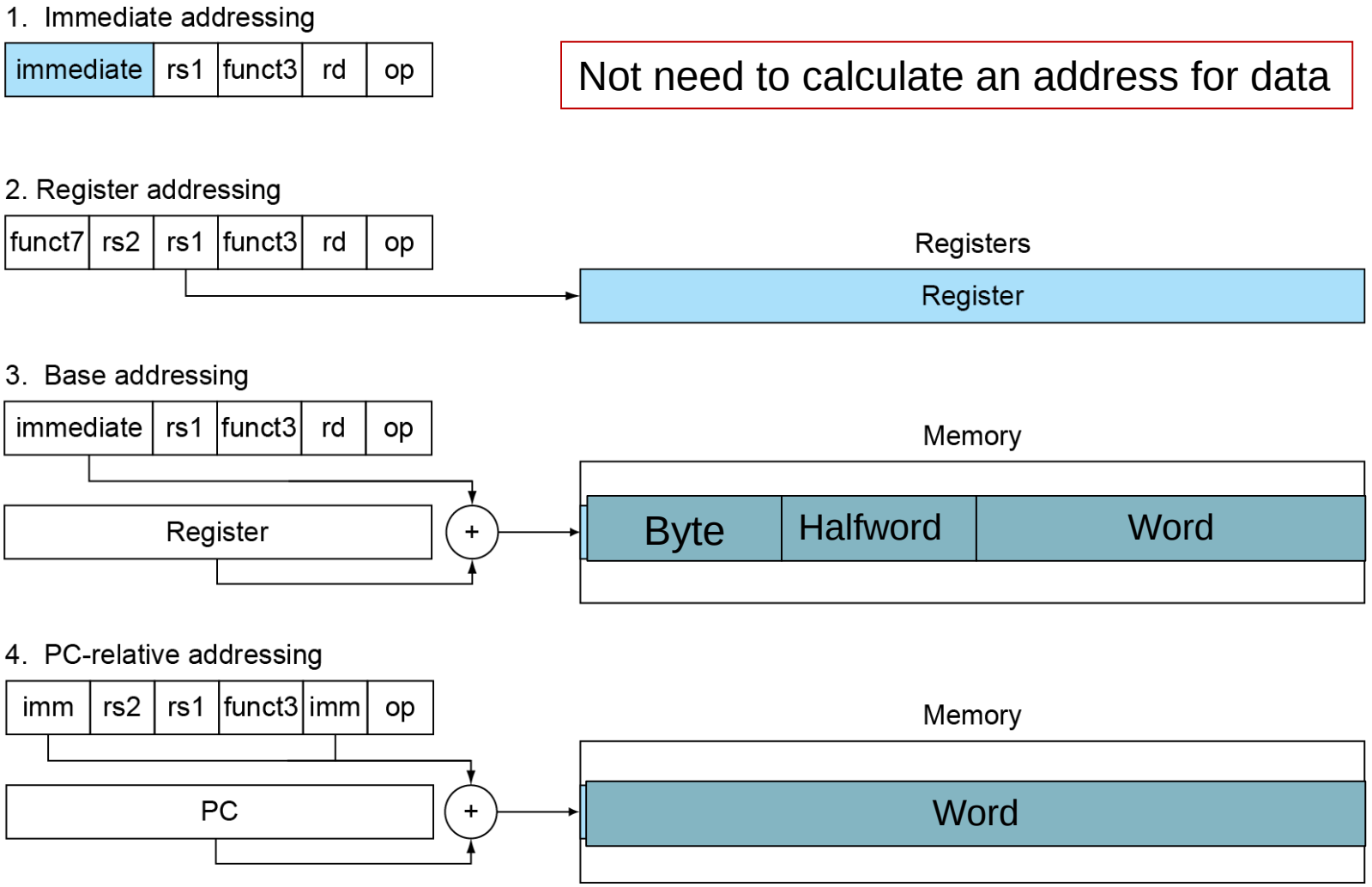
- Jump and link (`jal`) target uses 20-bit immediate for larger range
- **UJ** format:



- **U** format 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

Read also the Elaboration in page 121

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

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on order of accesses
- 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 (**used in RISC-V**)

Synchronization in RISC-V

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

Synchronization in RISC-V

- Example 1: atomic swapping a value in a register with the one in a memory location

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

```
sc.w x11,x23,(x20)
Read (x20), then store a “0” into x11 and store x23 into
(x20)if (x20) has not been changed.
Otherwise, store a “1” into x11.

So if (x20) initially has a value 5 and x23 has a value 1,
After doing lr.w and sc.w, if (x20) is not changed, x11
Will have a value 0. This means that the exchange is
successfully done. So, x10 (also x23) will have a value 5
and (x20) Will have a value 1.
```

Synchronization:

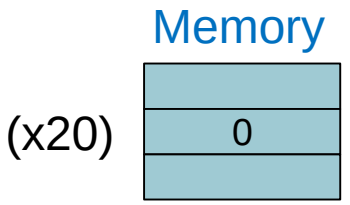
- Both processor A and B are running the same piece of code respectively. Assume (x20) initially has a value 0. Initially, x23 in both PA and PB has a value 1.

Processor A (PA)

```
again:  lr.w x10,(x20)
        sc.w x11,x23,(x20)
        bne  x11,x0,again
        addi x23,x10,0
```

Processor B (PB)

```
again:  lr.w x10,(x20)
        sc.w x11,x23,(x20)
        bne  x11,x0,again
        addi x23,x10,0
```



Initially, the lock is free

Assume PA and PB already executed **lr.w** and now executing **sc.w.** Assume PA finishes sc.w first and PB finishes it later. Then, PA will write a “0” to **its** x11 and store a “1” to (x20). So PA will get out of loop and copy its x10 to x23. On the contrary, PB will write a one to its x11 and continue performing the loop to test the lock.

Synchronization in RISC-V (cont.)

- Example 2: test/set lock variable (more efficient implementation)

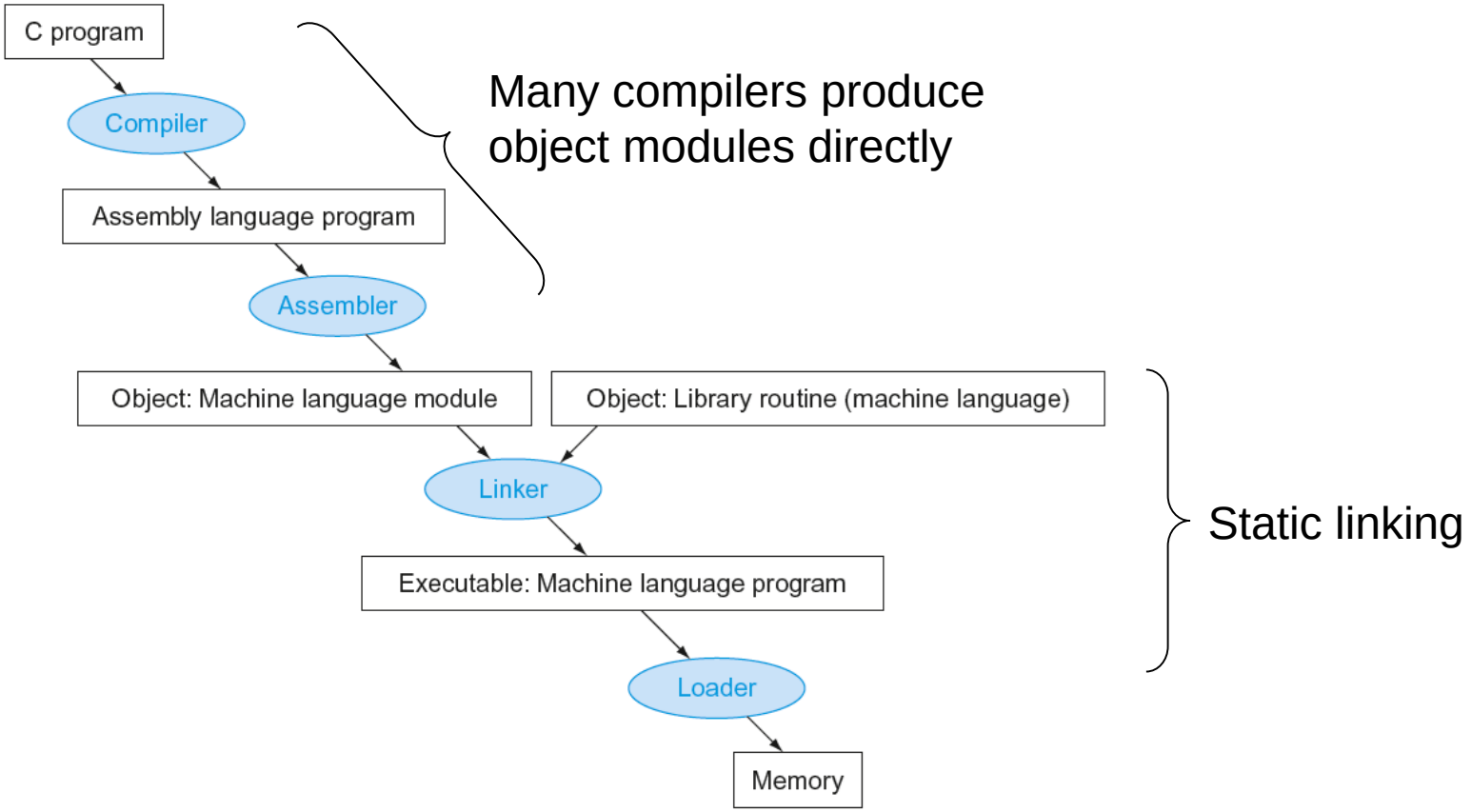
```
//A "1" stored in (x20) means the lock is set. A
  "0" means free.
```

```
        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,x12,(x20)     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- Unlock:

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

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

Study the example in page 135

- 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 an address space large enough for text and data
  3. Copy text and initialized data into memory
  4. Copy arguments of main(…) onto 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 loading the functions in the library being called but not actually executed.
  - Automatically picks up new library versions

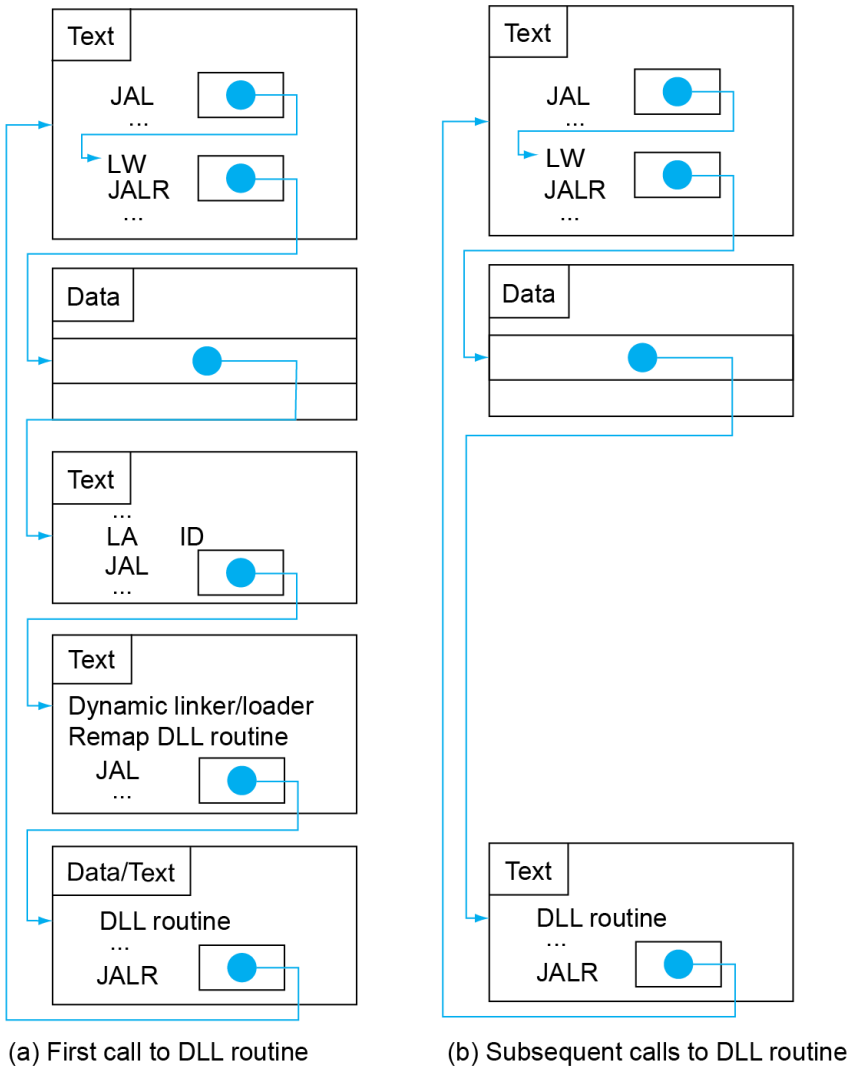
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

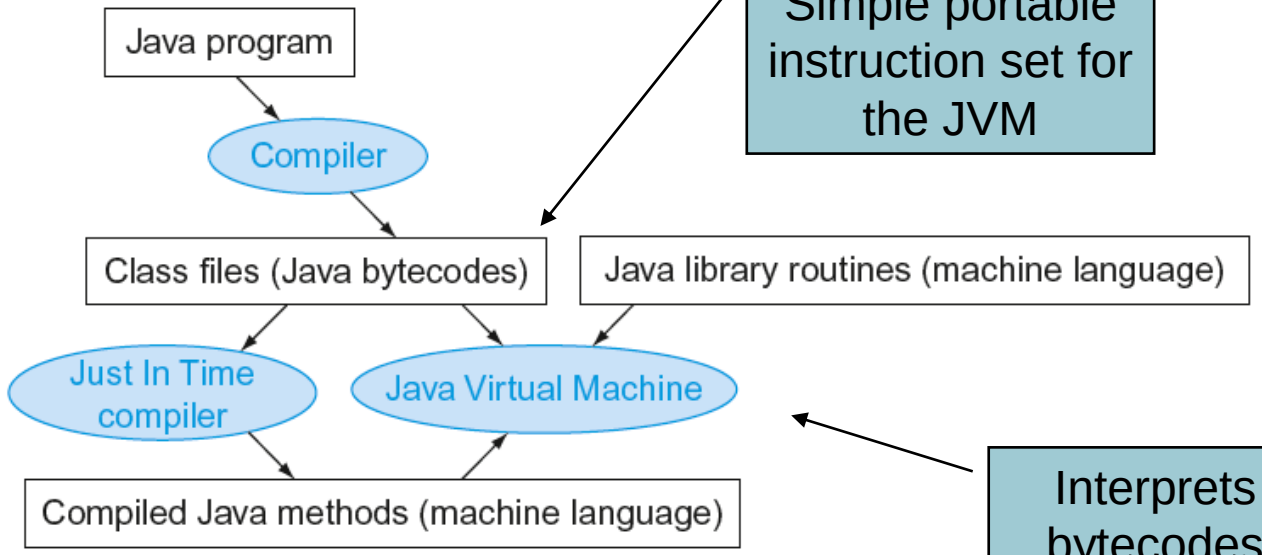
Dynamically mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Starting Java Applications



Simple portable instruction set for the JVM

Compiles bytecodes of "hot" methods into native code for host machine

Interprets bytecodes

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(long long int v[],
          long long int k)
{
  long long int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in x10, k in x11, temp in x5

The Procedure Swap

```
swap:
  slli x6,x11,2     // reg x6 = k * 4
  add  x6,x10,x6    // reg x6 = v + (k * 4)
  lw   x5,0(x6)     // reg x5 (temp) = v[k]
  lw   x7,4(x6)     // reg x7 = v[k + 1]
  sw   x7,0(x6)     // v[k] = reg x7
  sw   x5,4(x6)     // v[k+1] = reg x5 (temp)
  jalr x0,0(x1)     // return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], size_t n)
{
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   mv   x21,x10 //copy parameter x10 into x21 (base addr of v[])
   mv   x22,x11 // copy parameter x11 into x22 (array size n)
   li   x19,0       // i = 0  (addi x19, x0, 0)
for1tst:
   bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

   (body of inner for-loop)

   addi x19,x19,1         // i += 1
   jal  x0,for1tst        // branch to test of outer loop
exit1:
```

mv is a pseudo instruction.
mv x21, x10
is the same as
addi x21, x10, 0

The Inner Loop

- Skeleton of inner loop:
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {
    swap(v,j); }

```
    addi x20,x19,-1    // j = i –1
for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,2      // reg x5 = j * 4
    add  x5,x21,x5     // reg x5 = v + (j * 4)
    lw   x6,0(x5)      // reg x6 = v[j]
    lw   x7,4(x5)      // reg x7 = v[j + 1]
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
    mv   x10, x21      // first swap parameter is v
    mv   x11, x20      // second swap parameter is j
    jal  x1,swap          // call swap
    addi x20,x20,-1       // j -= 1
    jal  x0,for2tst       // branch to test of inner loop
exit2:
```

Preserving Registers

- Preserve saved registers:

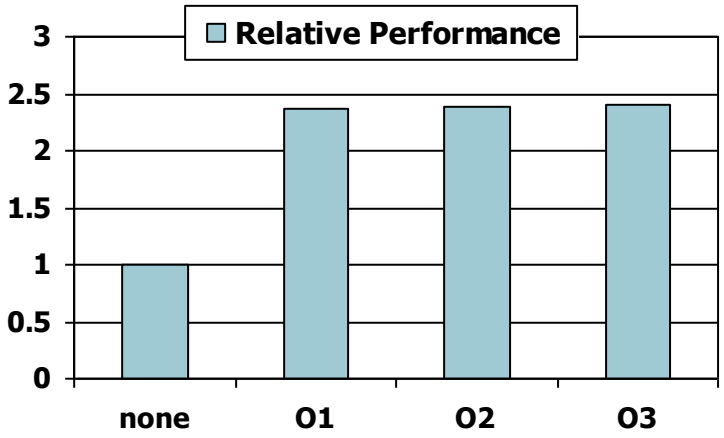
```
sort:
  addi sp,sp,-20  // make room on stack for 5 regs
  sw   x1,16(sp)  // save x1 on stack
  sw   x22,12(sp) // save x22 on stack
  sw   x21,8(sp) // save x21 on stack
  sw   x20,4(sp)  // save x20 on stack
  sw   x19,0(sp)  // save x19 on stack
```

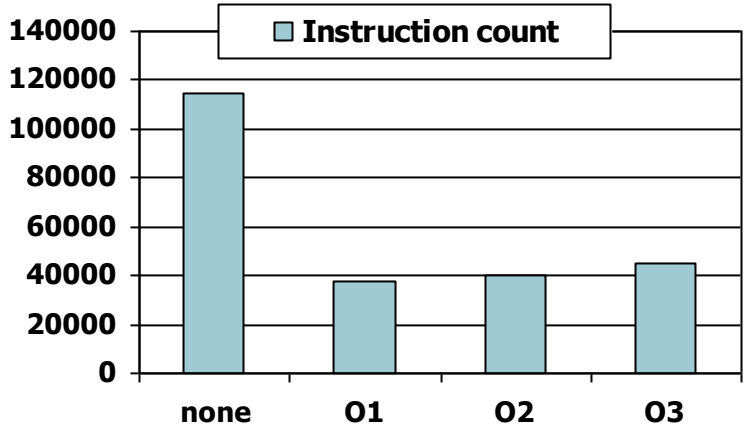
- Restore saved registers:

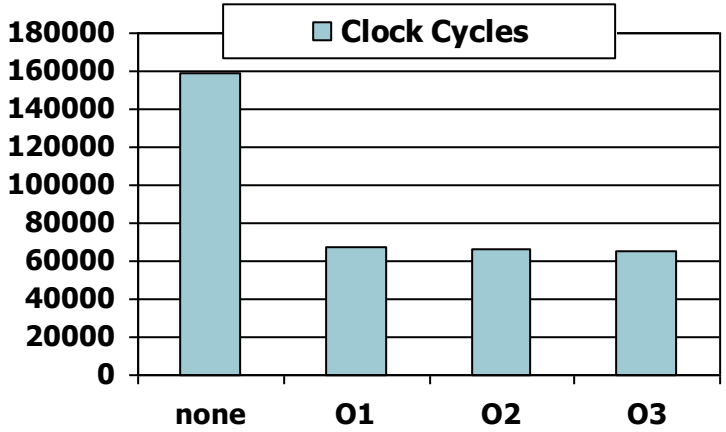
```
exit1:
   lw   x19,0(sp) // restore x19 from stack
   lw   x20,4(sp)  // restore x20 from stack
   lw   x21,8(sp) // restore x21 from stack
   lw   x22,12(sp) // restore x22 from stack
   lw   x1,16(sp)  // restore x1 from stack
   addi sp,sp, 20  // restore stack pointer
  jalr x0,0(x1)
```

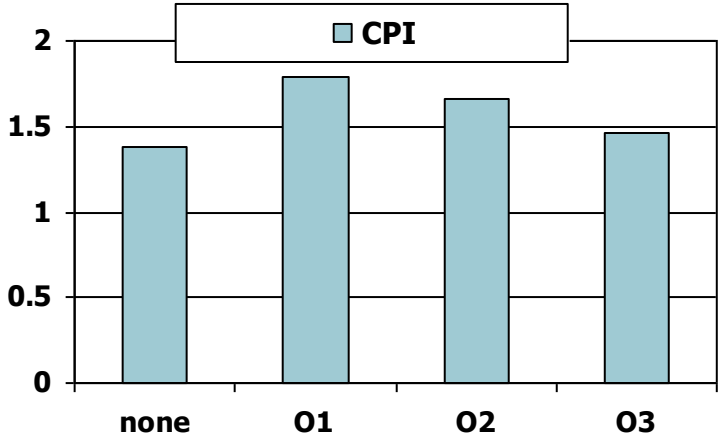
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

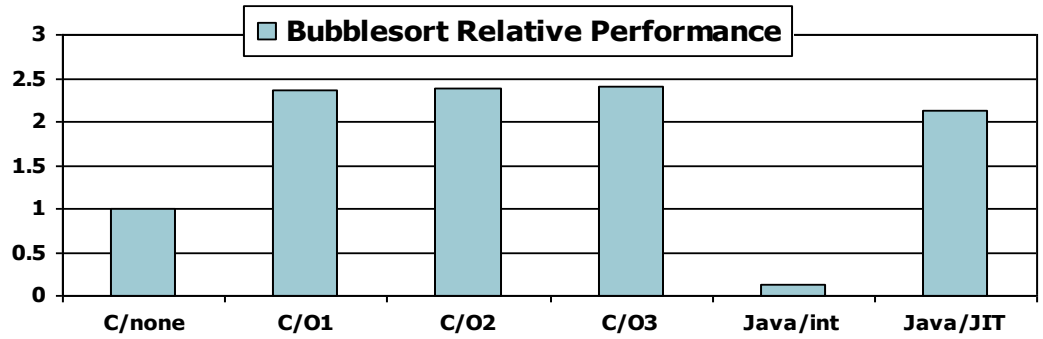


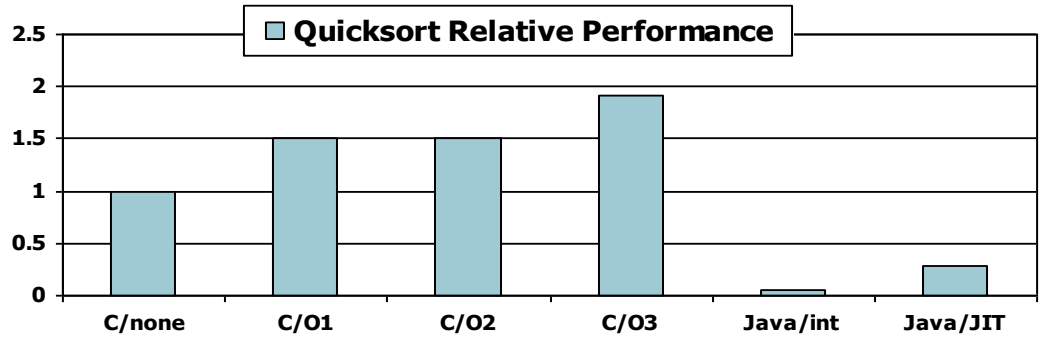


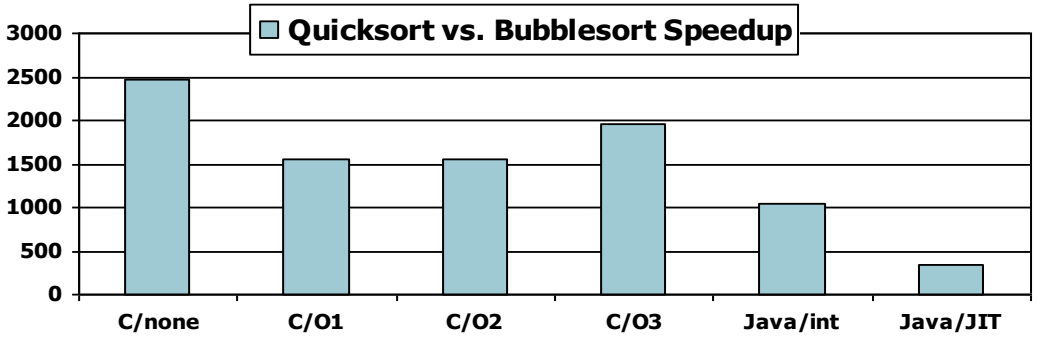




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

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;
}
```

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

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

```
   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

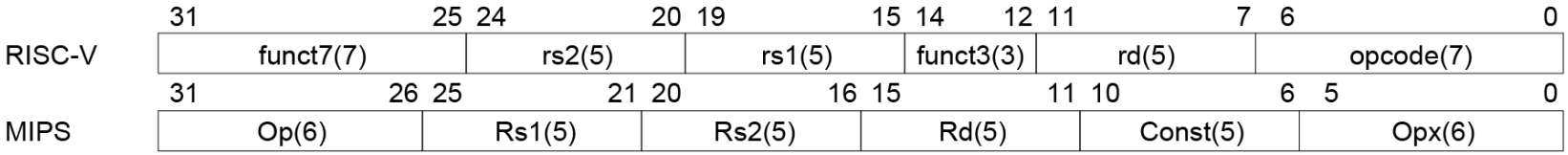
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and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

MIPS Instructions

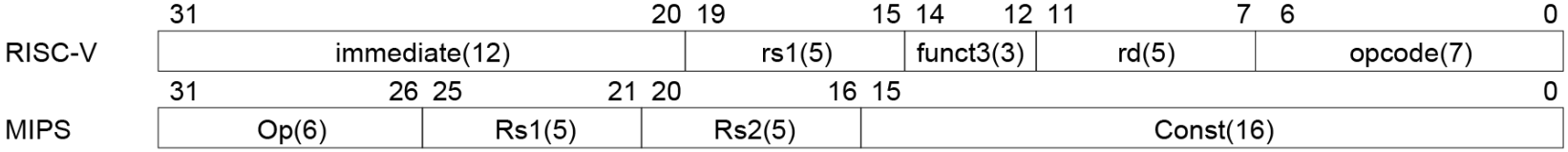
- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- Different conditional branches
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

Instruction Encoding

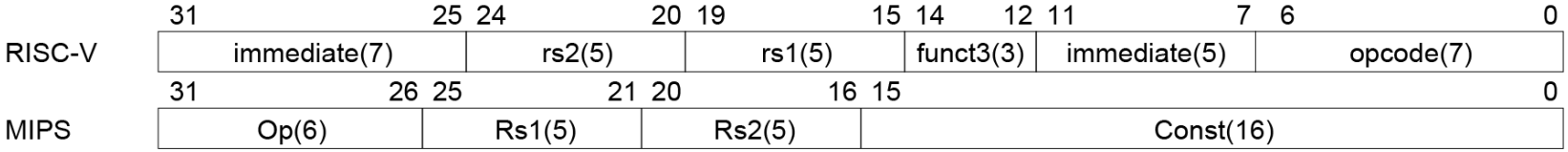
Register-register



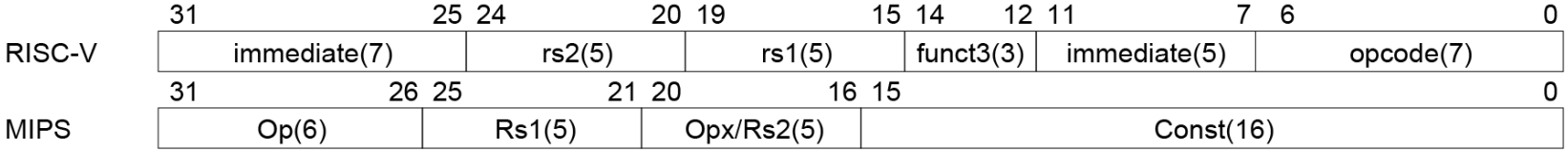
Load



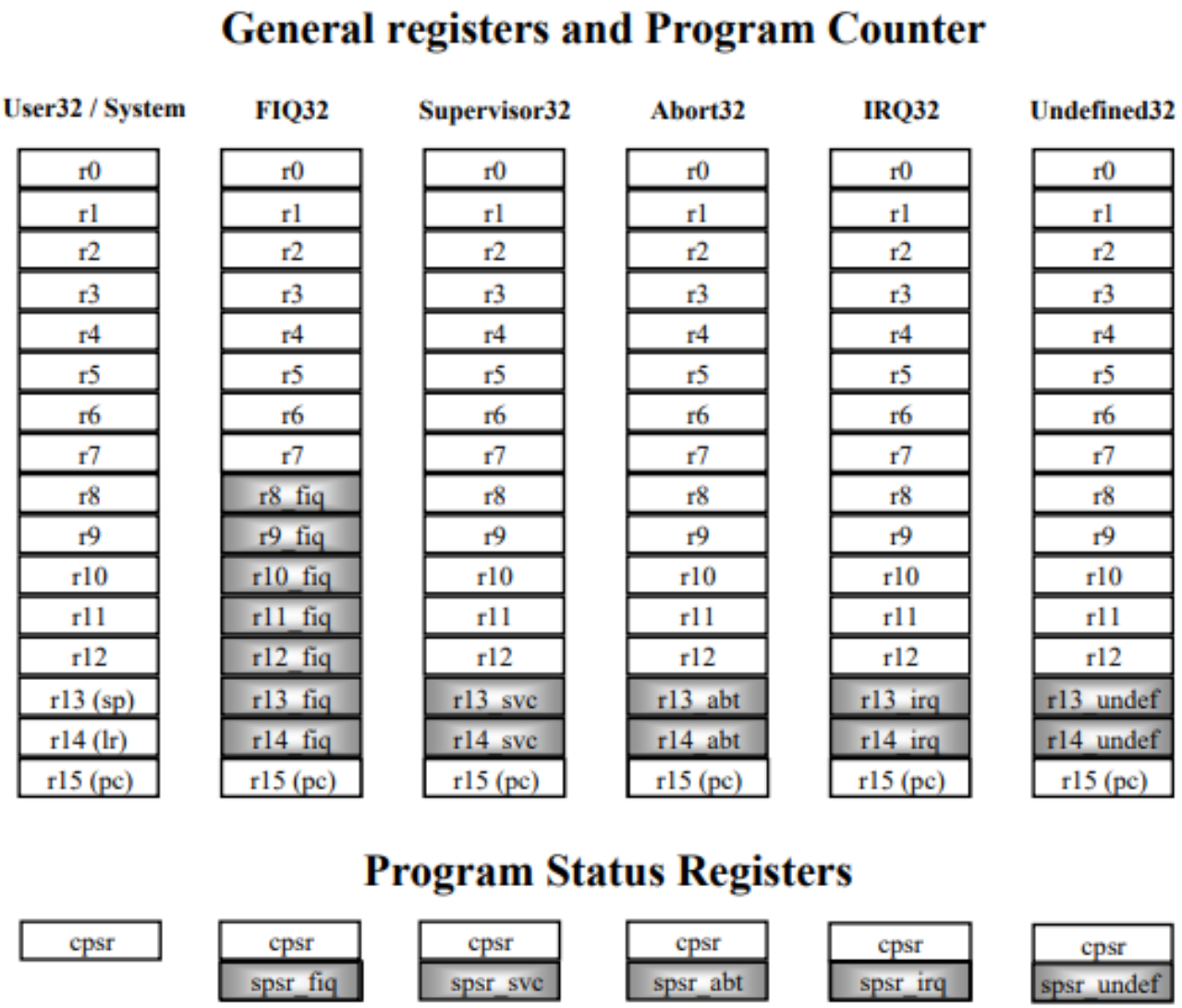
Store



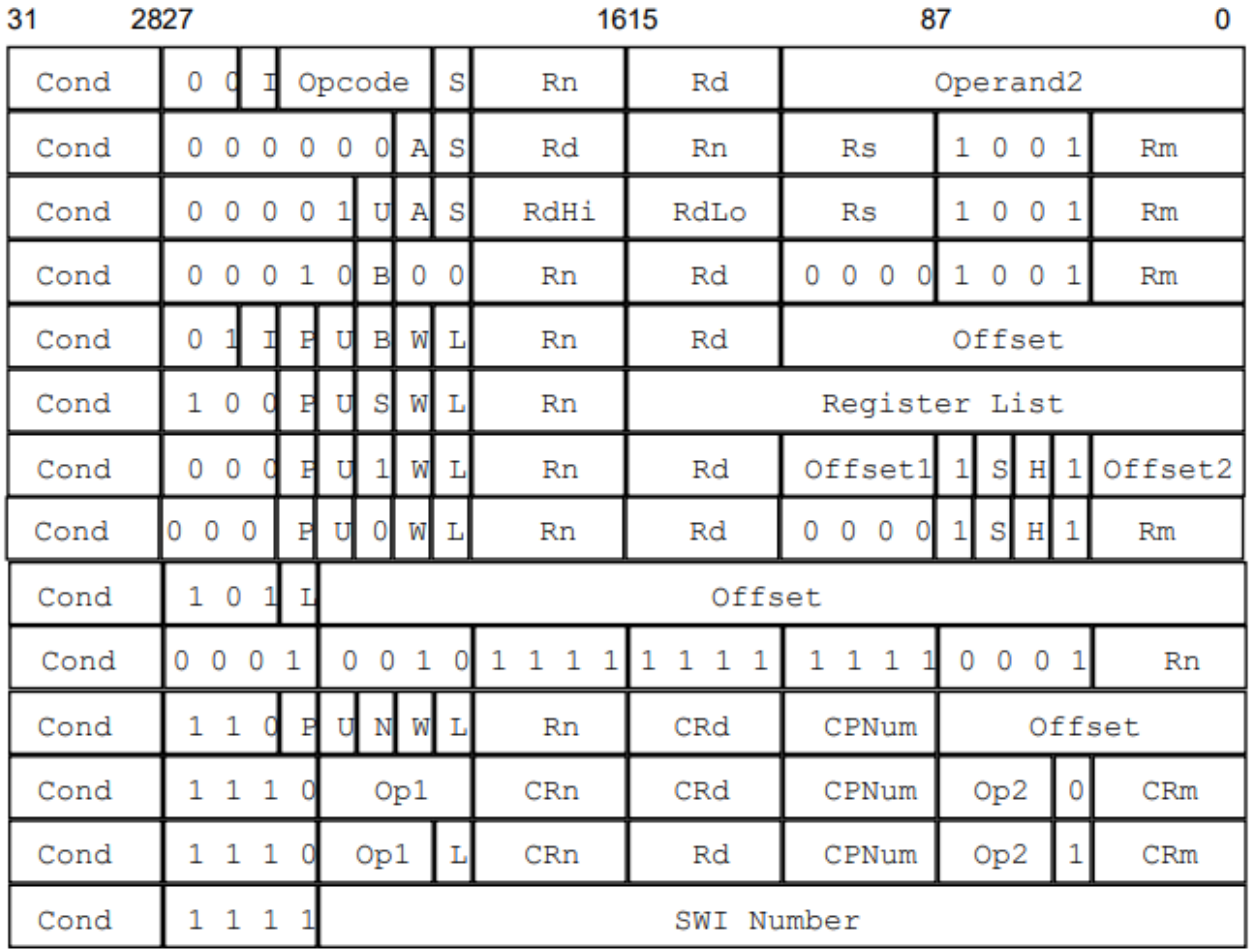
Branch



ARM ISA


General registers and Program Counter
User32 / System
FIQ32
Supervisor32
Abort32
IRQ32
Undefined32
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
r8_fiq
r9_fiq
r10_fiq
r11_fiq
r12_fiq
r13_fiq
r14_fiq
r13_svc
r14_svc
r13_abt
r14_abt
r13_irq
r14_irq
r13_undef
r14_undef
Program Status Registers
cpsr
spsr_fiq
spsr_svc
spsr_abt
spsr_irq
spsr_undef

ARM Instructions/Format



Instruction type

Data processing / PSR Transfer

Multiply

Long Multiply **(v3M / v4 only)**

Swap

Load/Store Byte/Word

Load/Store Multiple

Halfword transfer : Immediate offset **(v4 only)**

Halfword transfer: Register offset **(v4 only)**

Branch

Branch Exchange **(v4T only)**

Coprocessor data transfer

Coprocessor data operation

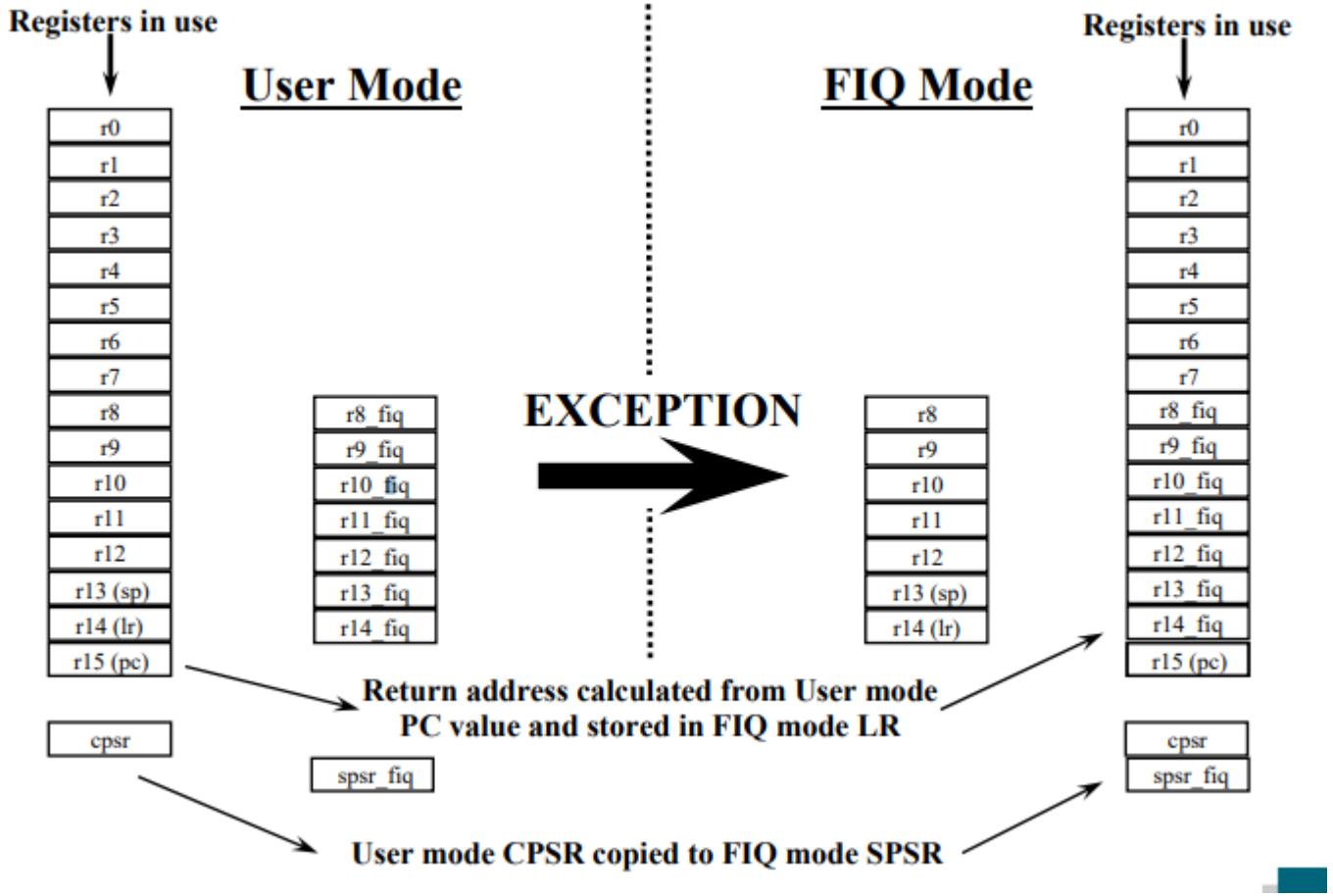
Coprocessor register transfer

Software interrupt

Ignore FIGURE 2.30, 2.31, 2.32, 2.33 in the textbook. These figures are wrong.

ARM's Mode Switch

- From user to Fiq (Fast interrupt) mode



The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

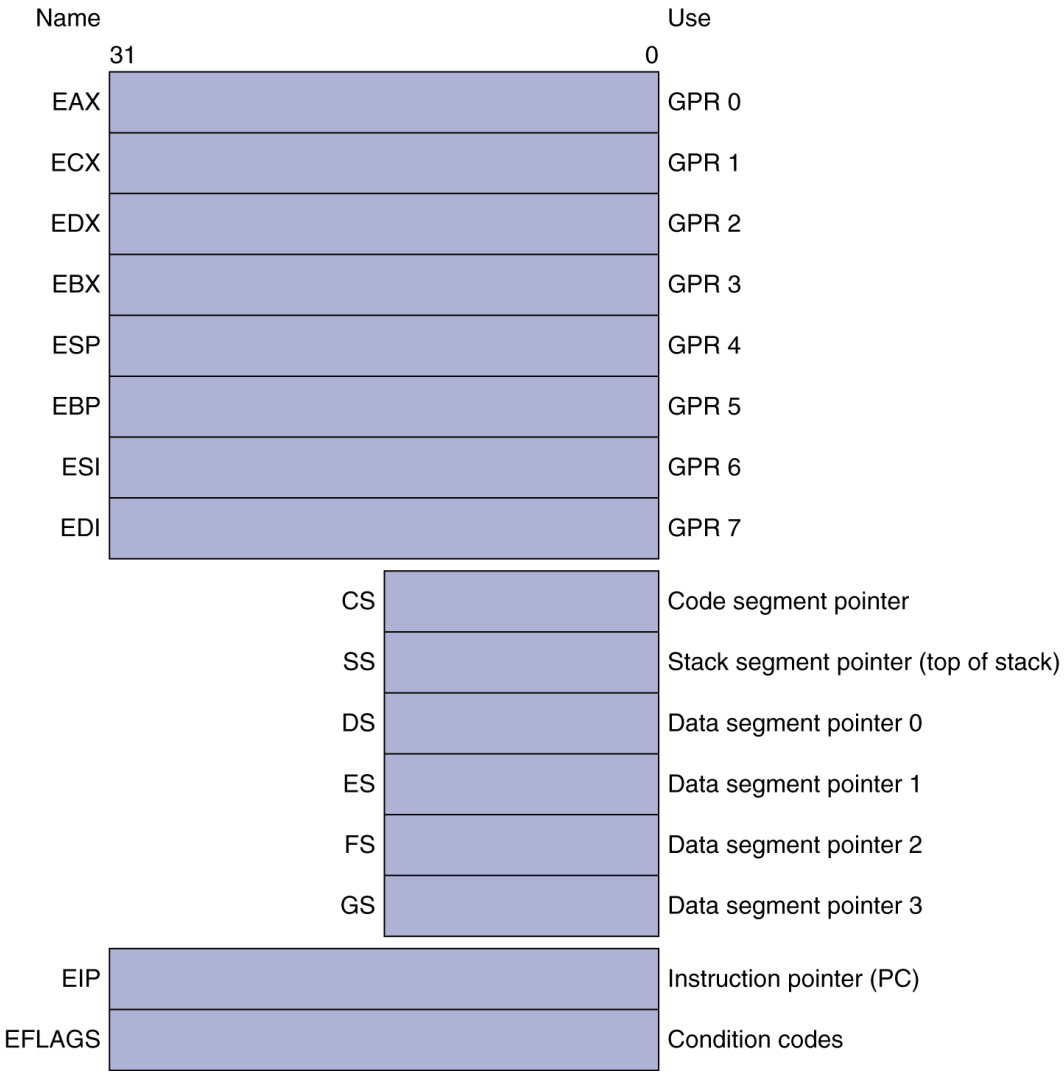
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
  - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers (32-bit AR.)



Basic x86 Addressing Modes

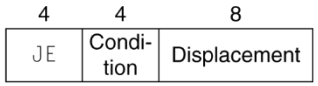
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

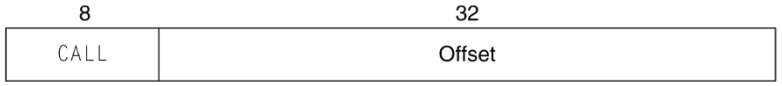
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

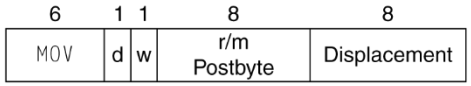
a. JE EIP + displacement



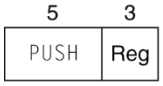
b. CALL



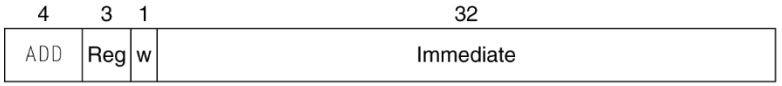
c. MOV EBX, [EDI + 45]



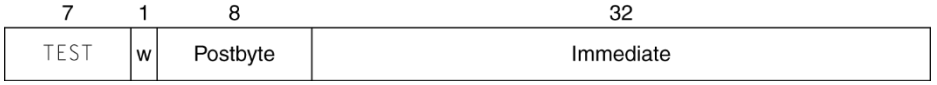
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1 **(one x86 inst corresponding to 1 microoperation)**
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - **auipc rd, immed** // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

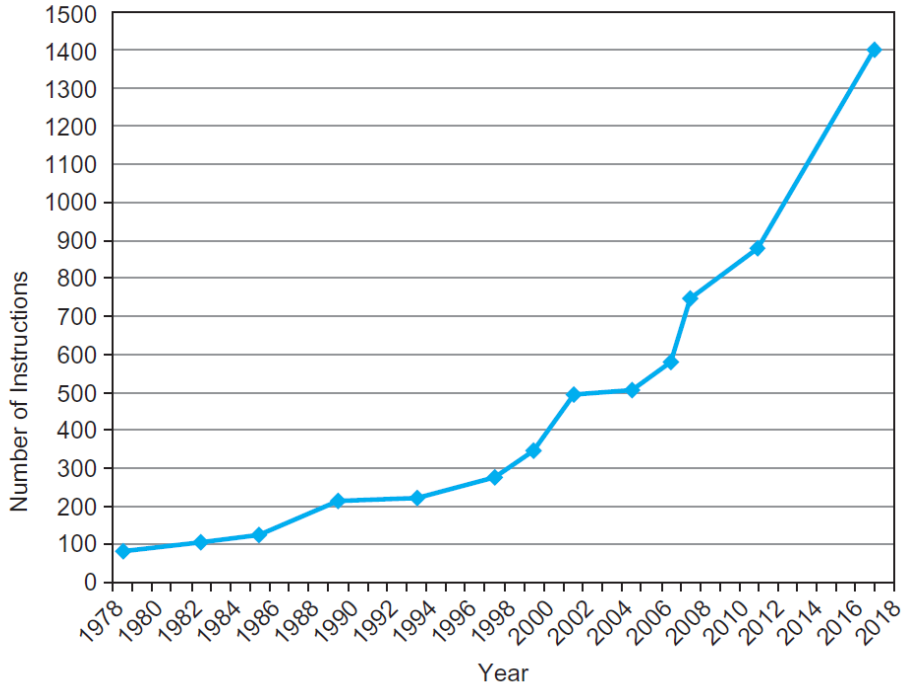
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped if the pointer is pointing to a local variable, esp. a local array.

Concluding Remarks

- Design principles
  1. Simplicity favors 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
  - c.f. x86