EECS4030: Computer Architecture

Instructions: Language of the Computer (II)

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

Outline

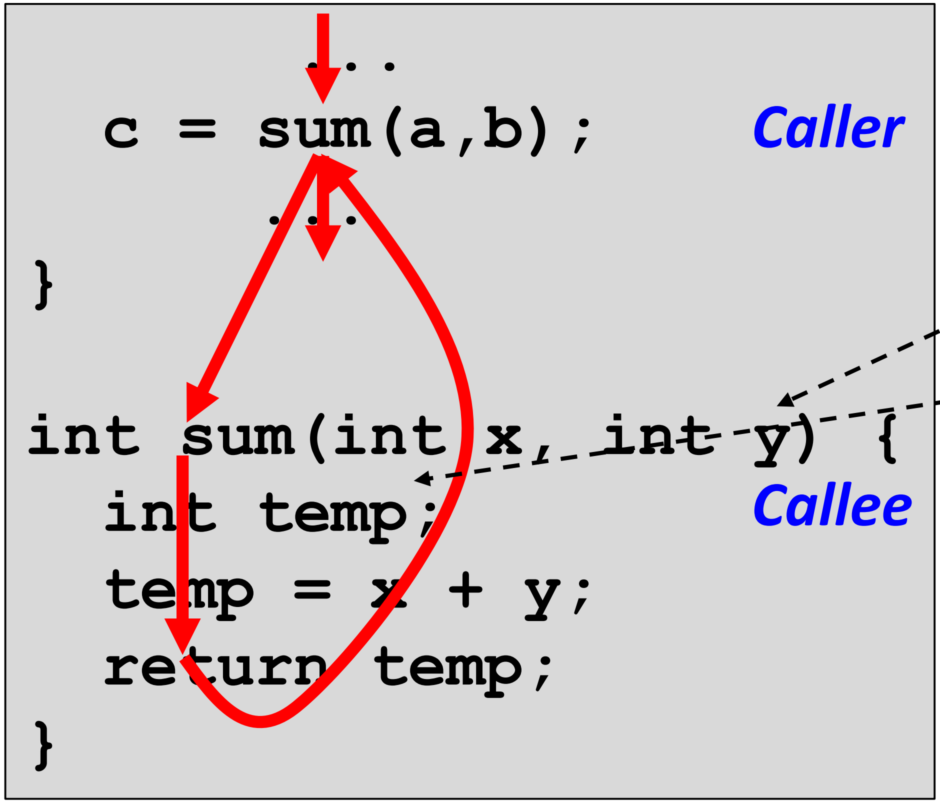
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Procedures/Functions in HLL

- Procedure/function calls change control flow
  - Procedures are self-contained and can be developed indep.



```
      ...
   c = sum(a,b);
     ...
}

int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
```

Must specify:

- Procedure address
- *Arguments* (or *parameters*)
- Local variables
- Return value
- Return address

How to implement it in machine code?

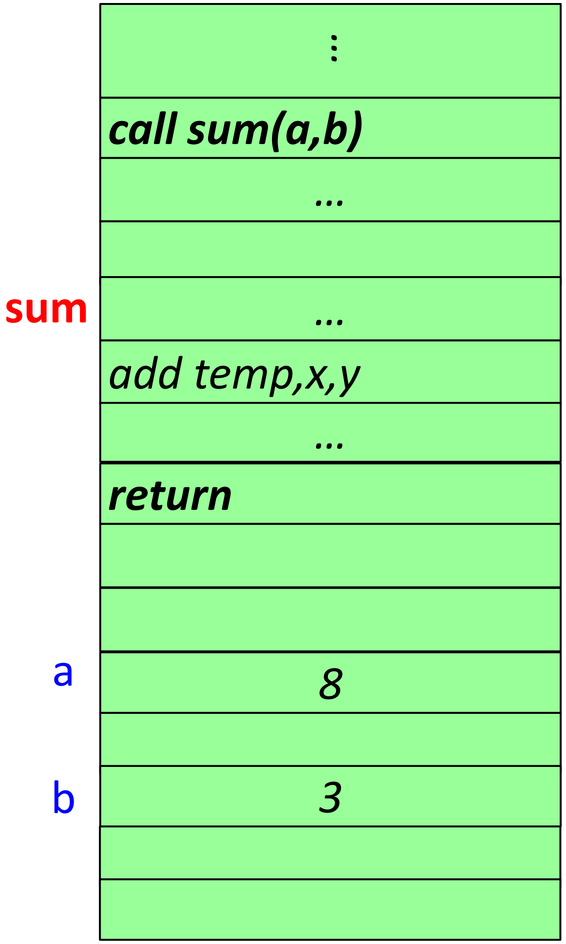
Note: for 64-bit RISC-V we should really use `long long int`

Program and Data Are Stored in Memory

- So, each instruction and data has an address
- Variable/procedure names ←→ memory address
  - The *label*, **sum**, in assembly code has a value corresponding to the beginning address of the procedure in memory

```
          ...
   c = sum(a,b);
          ...

int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
```



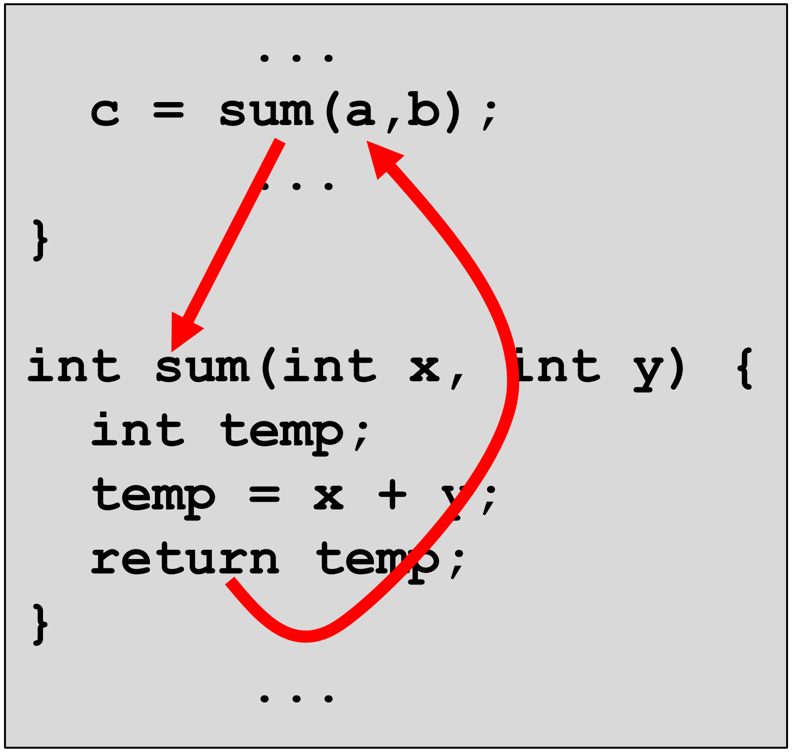
4 Questions for Implementing Procedures

- How to jump to and return from a procedure?
  - Procedure address, e.g. **`sum`**, and return address
  - RISC-V instructions to change the control flow
- How to pass data to and return data from procedure?
  - Arguments and return value
  - Note: C/C++ calls by values, not memory locations!
- Where to allocate local variables?
  - Note: C/C++ local variables are *automatic* variables
- How to handle recursive calls?
  - Procedure code is re-entered

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

1. Procedure Address and Instructions

- Procedure call:
  - Like a jump, but must remember where to return back → address of the next instruction after the call
  - Where to remember? → register ✓, memory
- Procedure return:
  - Jump back to execute from the address previously remembered
- Procedure address:
  - Allocated by assembler or programmer



```
          ...
   c = sum(a,b);
          ...
}

int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
          ...
```

RISC-V Procedure Call Instructions

Two procedure call instructions:

- *jump and link:* **`jal x1,ProcedureLabel`**
  - Puts address of following instruction (PC + 4) into **`x1`**
  - Jumps to location of **`ProcedureLabel`** by loading the 64-bit address into PC register
  - Why have a **`jal`**?
    Make the common case fast: functions are very common
- *jump and link register:* **`jalr x1,100(x20)`**
  - Like **`jal`**, but jumps to 100 + address in **`x20`**
  - Allows to jump to different procedures by changing the content in **`x20`**

RISC-V Procedure Return Instruction

- No special procedure return instruction in RISC-V
- Reuse *jump and link register*

```
    jalr x0,0(x1)
```

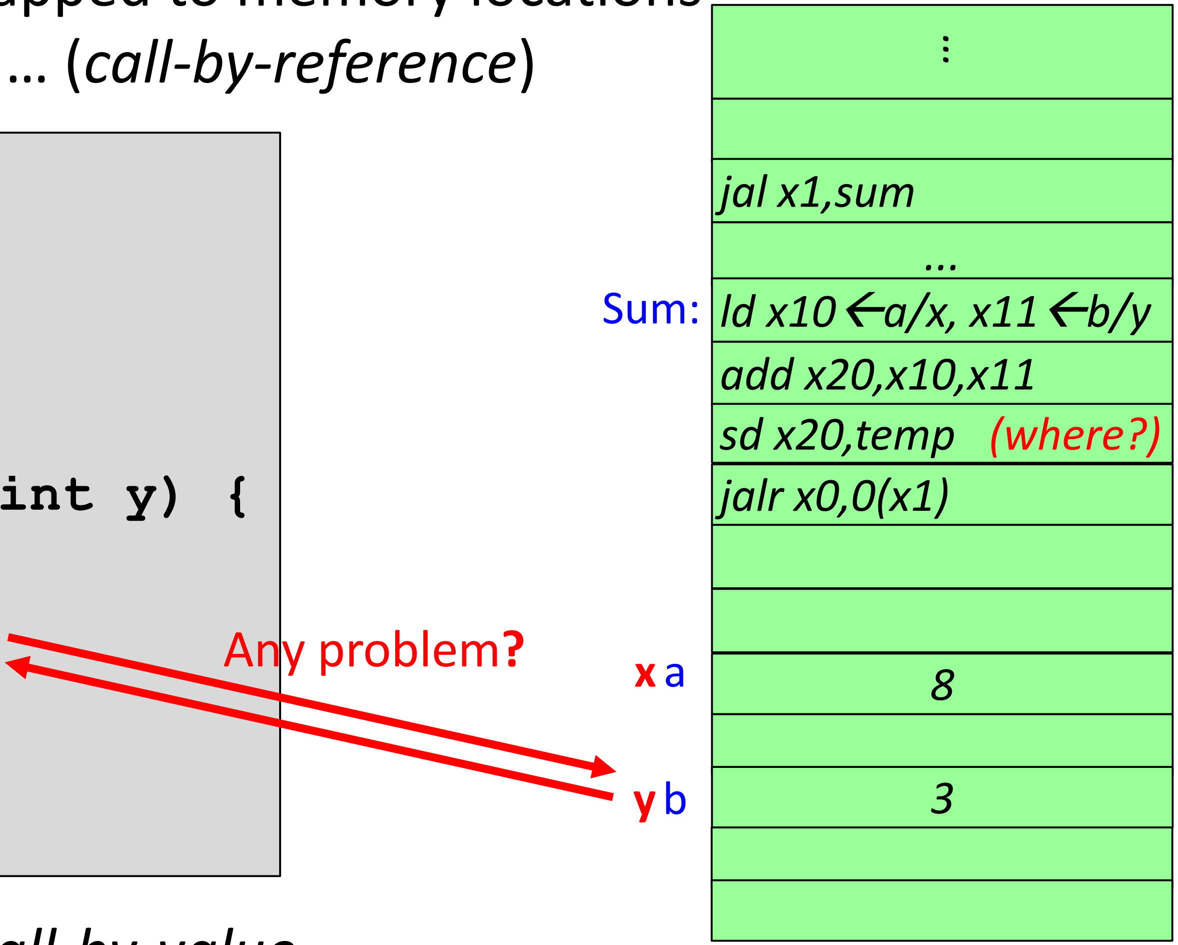
  - Assume `x1` contains the return address remembered when procedure was called: `jal x1,ProcedureLabel`
  - This instruction causes the processor to jump back to the return address
  - Use `x0` as `rd` (hardware still tries to put PC + 4 into `x0`, but `x0` cannot be changed)
  - Can also be used for computed jumps, e.g. case/switch, by loading the beginning address of different cases onto `x1`

2. Arguments and Return Value

- If arguments are mapped to memory locations of caller’s variables … (*call-by-reference*)

```
        . . .
   c = sum(a,b);
        . . .
}

int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
        . . .
```



⋮

jal x1,sum

…

Sum: *ld x10 ←a/x, x11 ←b/y*

add x20,x10,x11

sd x20,temp (where?)

jalr x0,0(x1)

x a *8*

y b *3*

Any problem**?**

C/C++ adopts *call-by-value*

2. Arguments and Return Value

- Call-by-value in C/C++

```
        ...
   c = sum(a,b);
        ...
}
             8      3
int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
        ...
```

Conceptually:

- Callee receives parameters as values (3 & 8) from caller
- Not knowing where caller stores the values (why?)
- It keeps the values in its local storage, x and y, that the caller needs not know

➔ **abstraction/encapsulation**

In practice:

- Need a common storage to pass values → registers

Note: Caller can willingly grant callee accesses to its storage by passing pointers (addr)

2. Arguments and Return Value

- Call-by-value in C/C++
  - Return value similar

e.g. use x10 and x11 to pass parameters (by compiler)

```
        ...
  c = sum(a,b);
        ...
}
                8         3
                ↓         ↓
int sum(int x, int y) {
  int temp;
  temp = x + y;
  return temp;
}
        ...
```

```
     ld   x10,0(x5)  // x10←M[a]
     ld   x11,0(x6)  // x11←M[b]
     jal  x1,sum     // call sum

     ...
sum:

     add  x20,x10,x11 // x + y
     ...
```

Need calling convention!

Note: What if registers are not enough to hold all arguments?

3. Local Automatic Variables

- Automatic variables:
  - Allocated and deallocated automatically when control flow enters and leaves the variable's scope (Wikipedia)
  - Inaccessible outside the scope
- Where to allocate?
  - Registers: e.g. x20
    - Must save old contents → where to?
    - Not enough registers?
  - Fixed memory location:
    - Similar to static variables
    - What’s the problem?
  - Relative memory location:
    - Relative to what?

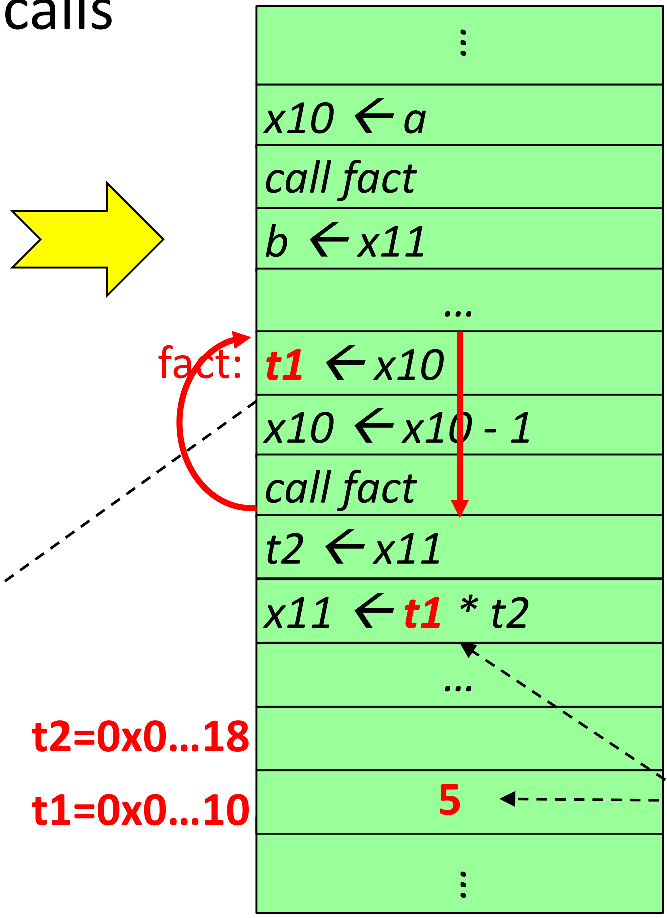
```
          ...
   c = sum(a,b);
          ...
}

int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp;
}
          ...
```

3. Local Automatic Variables

- Why can’t assign automatic variables fixed locations?
  - Consider recursive calls

```
   ...
b = fact(a);
   ...
fact(int n){
 int t1=n,t2;
 t2 = fact(n-1);
 return t1*t2;
}
```



⋮

x10 ← a

call fact

b ← x11

…

fact:

t1 ← x10

x10 ← x10 - 1

call fact

t2 ← x11

x11 ← t1 * t2

…

t2=0x0…18

t1=0x0…10

5

⋮

M[a] = 5

t1 = ?

t1 is accessible to and reused by all instances of fact()

Need to remember 5 so as to calculate 5! after recursive call returns

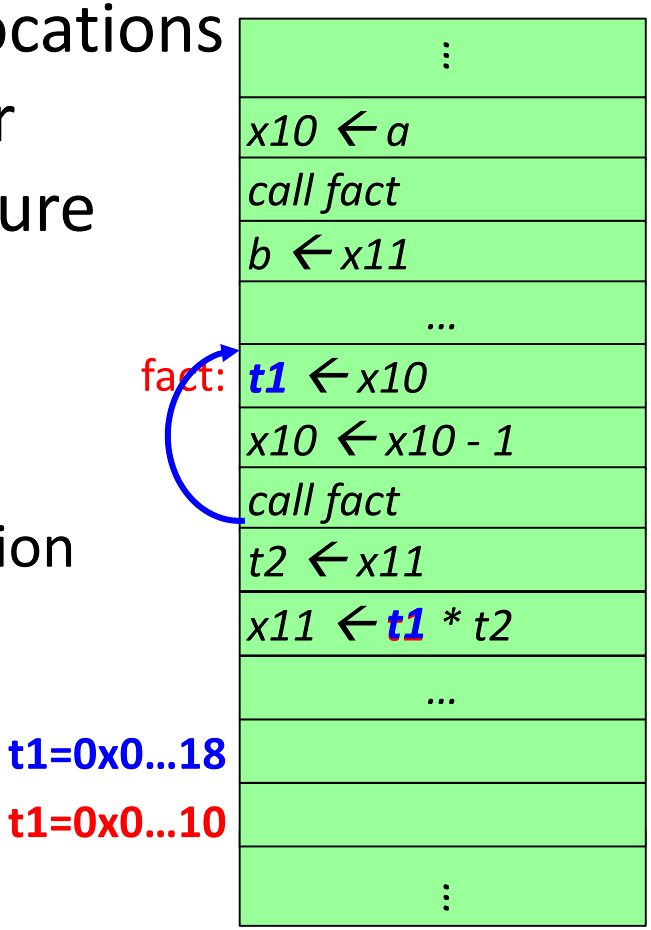
```
//x20←0x0…10
sd   x10,0(x20)
addi x10,x10,-1
...
```

3. Local Automatic Variables

- What we want is (1) same sequence of instructions that (2) access different memory locations for the same automatic variable for different invocations of the procedure

- How can it be done?
  - Base addressing with a base register
  - Change base register on each invocation

```
addi  x2,x2,8
sd    x10,0(x2)
addi  x10,x10,-1
...
```

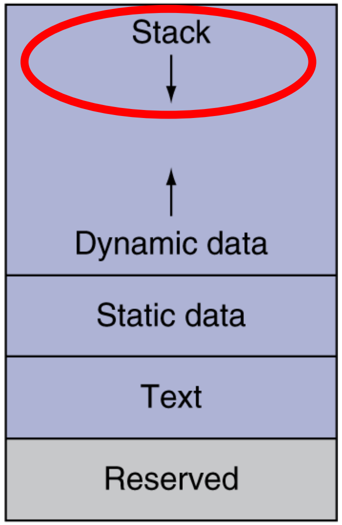


3. Local Automatic Variables

- What we don’t want is to mix this memory space with other static variables, e.g. global variables, because
  - We do not know how many levels the recursion can go and thus how large a space to reserve for automatic variables → not enough space may overwrite other variables
  - Also, local variables may be exposed to other procedures
- Live procedures are called first-in-last-out, so a nature way of allocating and managing that space is a separate **stack** space in memory

```
addi  sp,sp,-8
sd    x10,0(sp)
addi  x10,x10,-1
...
```

sp: *stack pointer*



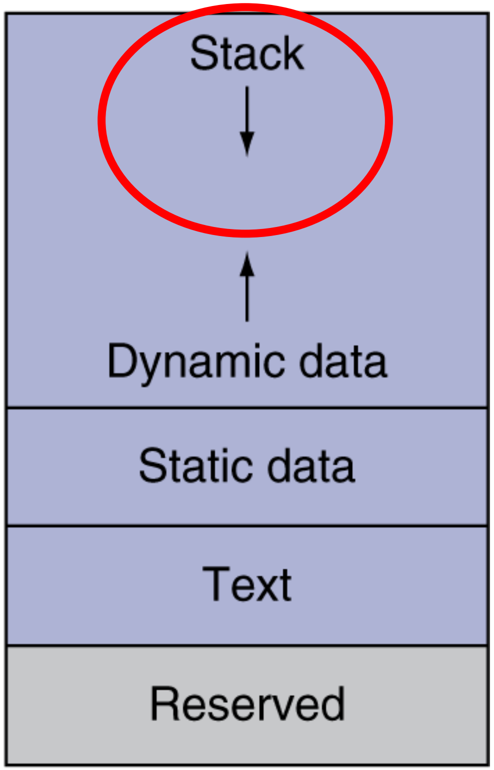
“Working Space” of “Live” Procedures

- Every “live” procedure/function must have its own working space (*frame*), i.e. memory space for all the variables that it needs, e.g. overflown arguments, saved register contents, automatic variables, etc.
  - Compiler knows size and contents of working space
  - Whenever a procedure is called, compiler generates some machine code to allocate this working space in stack
  - All local variables will find a location in the stack, addressed using SP (*stack pointer* register) as the base register
  - Different invocations of a procedure can run using the same code but work on different working space from the stack → ensuring abstraction and encapsulation

Procedures/Functions in HLL

- General strategies:
  - Compiler allocates all memory required of a procedure in *program stack* (pointed to by *stack pointer* register SP)
    - Typical memory layout seen by each program (compiler generates code to allocate and manage the memory space)
  - Use registers to transfer addresses and data as much as possible

```
c = sum(a,b);
...
int sum(int x, int y) {
   int temp;
   temp = x + y;
   return temp; }
```



Summary: Procedure Call in RISC-V

- **Caller:**

1) Place arguments in registers (x10 to x17)

2) Jump to beginning of procedure, save next PC in x1

- **Callee:**

3) Allocate storage on program stack

4) Perform procedure's operations

5) Place result in register (x10) for caller

6) Return to place of call at caller

- **Caller:**

7) Get return result from register (x10)

Caller and callee use registers to communicate as much as possible

Also need to backup and restore registers so that caller will not know

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`**, **`h`**, **`i`**, **`j`** in **`x10`** ~ **`x13`**
  - *Local variable*: **`f`** in **`x20`** (assigned by the compiler)
  - *Temporaries*: **`x5`**, **`x6`** (assigned by the compiler)
  - *Return value*: in **`x10`**
  - Need to save **`x5`**, **`x6`**, **`x20`** on stack in case caller uses them

Leaf Procedure Example

- RISC-V code:

Assembler changes sp to x2

```
leaf_example: addi sp,sp,-24
              sd   x5,16(sp)      save x5, x6, x20 on stack
              sd   x6,8(sp)
              sd   x20,0(sp)
              add  x5,x10,x11     x5 = g + h
              add  x6,x12,x13     x6 = i + j
              sub  x20,x5,x6      f = x5 – x6
              addi x10,x20,0      copy f to return register
              ld   x20,0(sp)
              ld   x6,8(sp)       retsore x5, x6, x20 from stack
              ld   x5,16(sp)
              addi sp,sp,24
              jalr x0,0(x1)       return to caller
```

Preserving Register Contents on the Stack

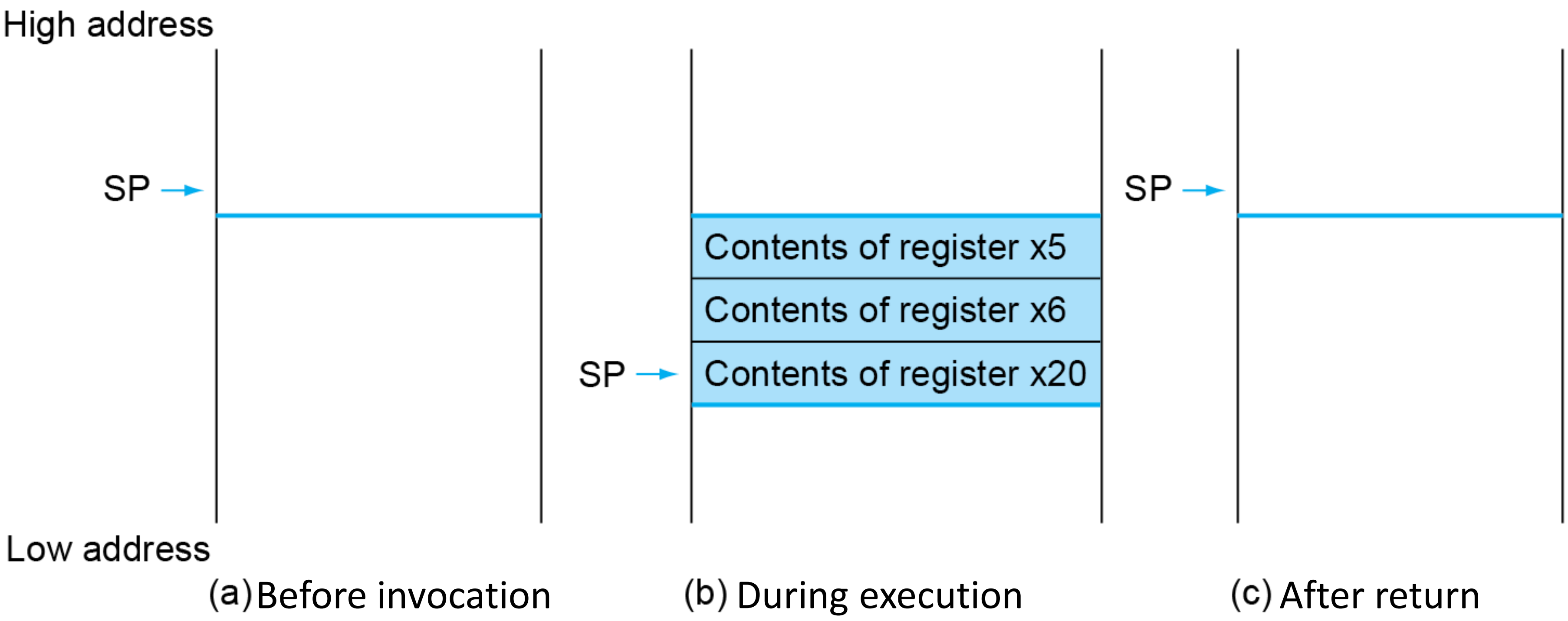


Fig. 2.10

Space allocated specifically for this invocation of the procedure

RISC-V Procedure Call Convention

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	---
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	---
x4	tp	Thread pointer	---
x5	t0	Temporary/alternate link register	Caller
x6~x7	t1~t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10~x11	a0~a1	Function arguments/return values	Caller
x12~x17	a2~a7	Function arguments	Caller
x18~x27	s2~s11	Saved registers	Callee
x28~x31	t3~t6	Temporaries	Caller

Fig. 2.14

(RISC-V Instruction Set Manual)

Procedure Call Convention

```
... sum(a,b)...;

long long int sum(long long int x,
                  long long int y) {
    long long int temp;
    temp = x + y;
    return temp;
}
```

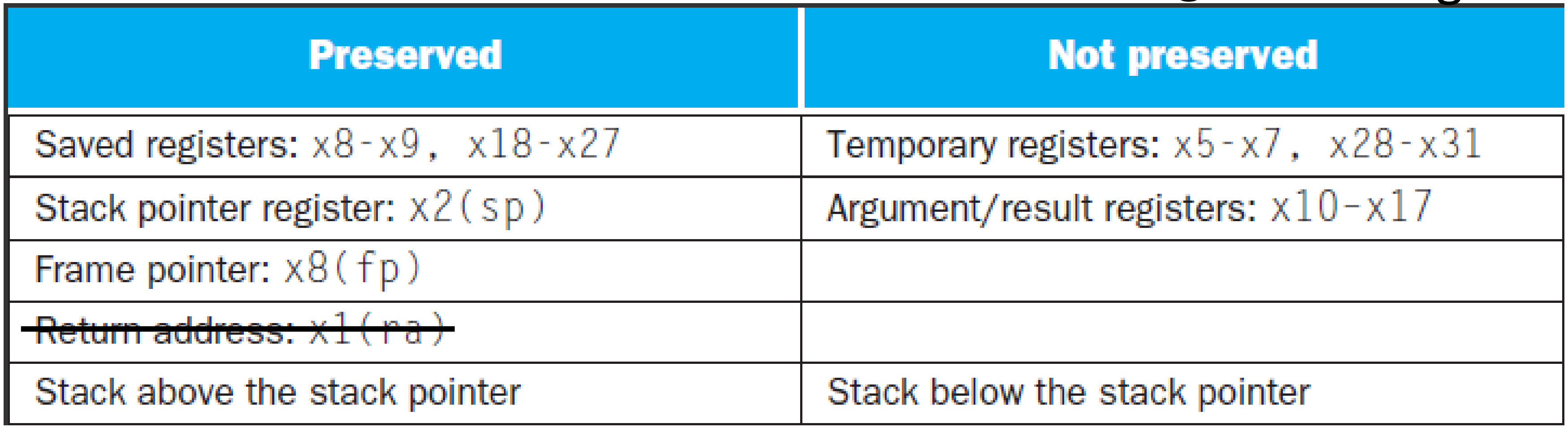
Return address	ra (x1)
Procedure address	Labels
Arguments	a0~a7 (x10~x17)
Local variables	t0~t6 (x5~x7,x28~x31)
Return value	a0,a1 (x10,x11)

Why such convention?

Why Procedure Convention Important?

- As a *contract* between caller and callee, so that
  - People who have never seen or even communicated with each other can write functions that work together

Fig. 2.11



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

  - Preserved: if used, callee saves and restores them in stack
  - Not preserved: callee uses them freely without preserving
    - So if caller needs them after the call, it has to preserve them

Based on this convention, x5, x6 in leaf procedure example need not be saved

Non-Leaf Procedures

- Procedures that call other procedures
- For nested calls, 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
- Example C code:

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

  - Argument **n** in **a0 (x10)**
  - Result value in **a0 (x10)**

Non-Leaf Procedure Example (Use ABI)

```
fact: addi sp,sp,-16      // make space on stack
      sd   ra,8(sp)       // save return address in x1 onto stack
      sd   a0,0(sp)       // save argument in x10 onto stack
      addi t0,a0,-1       // t0 = n – 1
      bge  t0,zero,L1     // if n >= 1, go to L1
      addi a0,zero,1      // else, set return value to 1
      addi sp,sp,16       // pop stack, don’t bother restoring values
      jalr zero,0(ra)     // return
L1:   addi a0,a0,-1       // n = n – 1
      jal  ra,fact        // call fact(n – 1)
      addi t1,a0,0        // move return value of fact(n – 1) to t1
      ld   a0,0(sp)       // restore caller’s n
      ld   ra,8(sp)       // restore return address
      addi sp,sp,16       // return space on stack
      mul  a0,a0,t1       // return n * fact(n – 1)
      jalr zero,0(ra)     // return
```

Non-Leaf Procedure Example (RISC-V Code)

```
fact: addi x2,x2,-16    // make space on stack
      sd   x1,8(x2)     // save return address in x1 onto stack
      sd   x10,0(x2)    // save argument in x10 onto stack
      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 x2,x2,16     // 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 return value of fact(n – 1) to x6
      ld   x10,0(x2)    // restore caller’s n
      ld   x1,8(x2)     // restore return address
      addi x2,x2,16     // return space on stack
      mul  x10,x10,x6   // return n * fact(n – 1)
      jalr x0,0(x1)     // return
```

Typical Memory Layout of a Program

Compilers allocate (map) variables to memory based on this (logical) view for each and every program

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

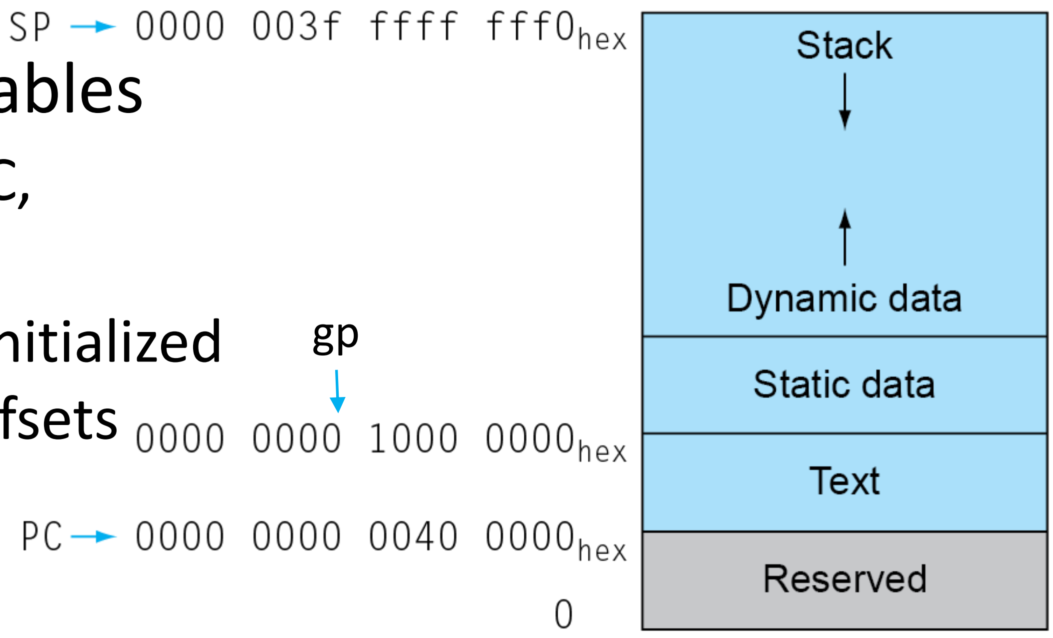
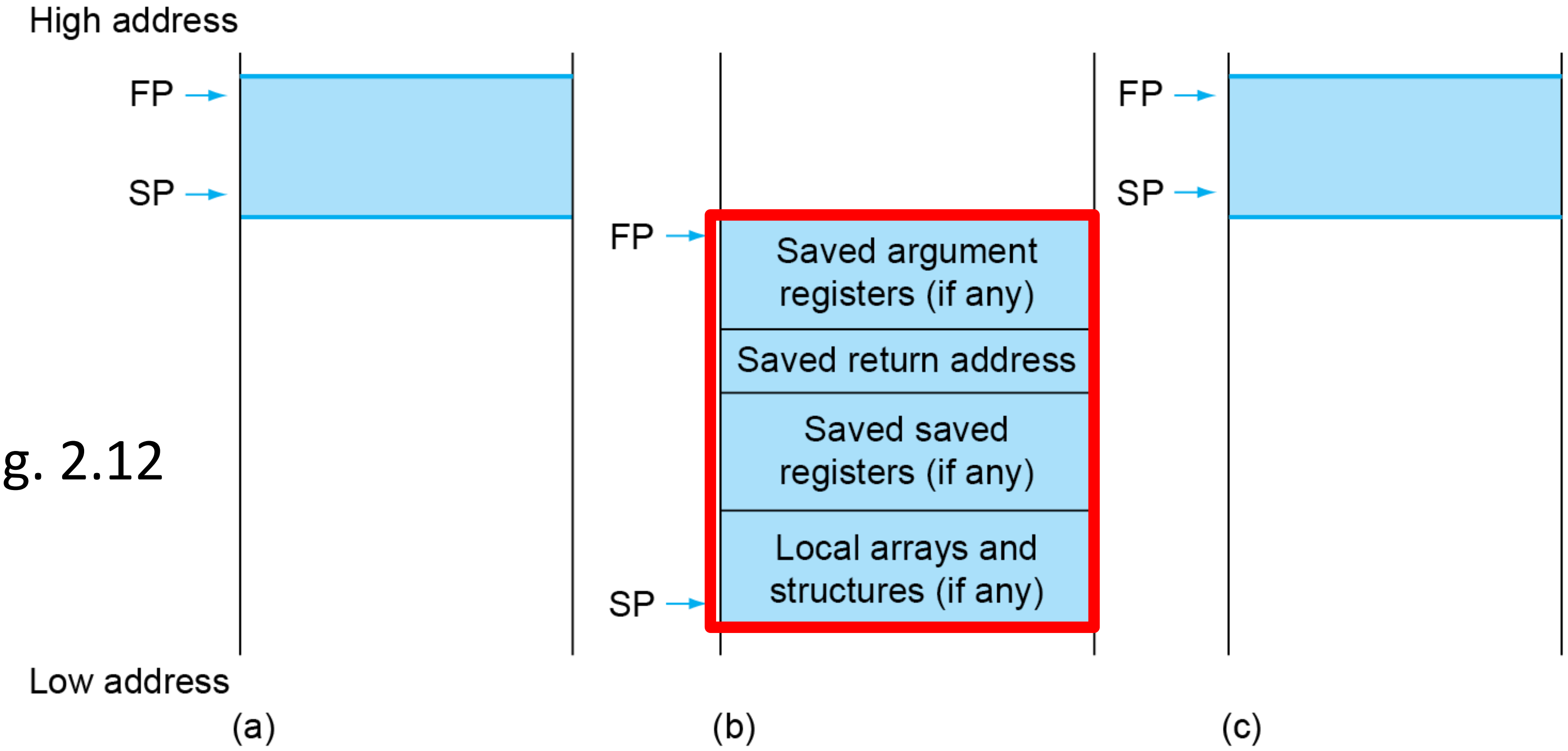


Fig. 2.13

Local Data on the Stack



High address
FP
SP
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
FP
SP
Low address
(a)
(b)
(c)

Fig. 2.12

- Local data allocated by callee, e.g., local array, in stack
- Procedure frame (*activation record*): **`fp (x8)`**
  - "Working space" of the procedure, compiler generated

Summary: Procedure Calls

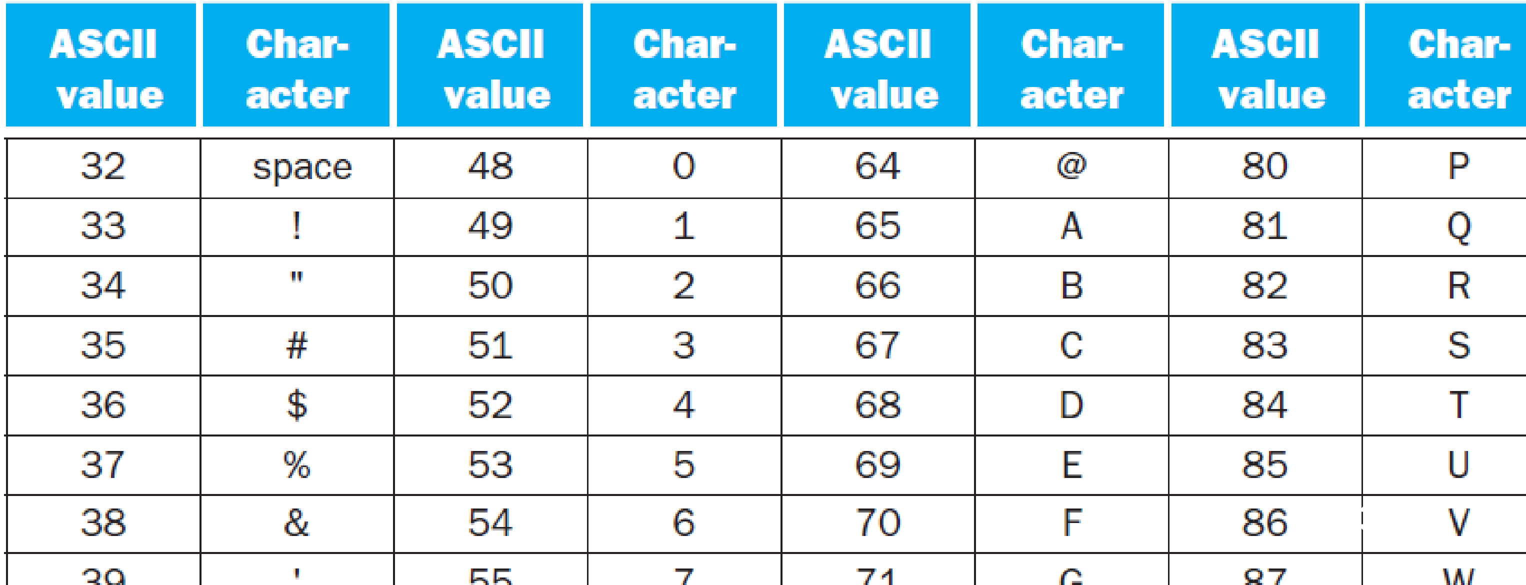
- Compiler (or assembly programmer) and processor hardware work together to support/translate procedure calls in HLLs
- Processor hardware provides:
  - Registers: SP (stack pointer), RA (return address), …
  - Instructions: **`jal`**, **`jalr`**
- Compiler does
  - Allocation of memory space for stack and local variables
  - Generation of instructions for managing stack, passing arguments and return values, jumping to and returning from procedure

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

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



ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P
33	!	49	1	65	A	81	Q
34	"	50	2	66	B	82	R
35	#	51	3	67	C	83	S
36	$	52	4	68	D	84	T
37	%	53	5	69	E	85	U
38	&	54	6	70	F	86	V
39	'	55	7	71	G	87	W

Fig. 2.15

RISC-V 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
    - **`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 for Handling Bytes

- C code:
  - Null-terminated strings

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

  - Addresses of **x**, **y** in **x10**, **x11** (argument registers)
  - **i** in **x19** (saved register → need to be preserved)
    - Compiler use registers x5~x7, x28~x31 for temporaries

String Copy Example

- RISC-V code: (pointer version)

```
strcpy:
    addi sp,sp,-8     // make space on stack
    sd   x19,0(sp)    // save x19
    add  x19,x0,x0    // x19 = 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     // exit if y[i] == 0
    addi x19,x19,1    // i = i + 1
    jal  x0,L1        // next iteration of loop
L2: ld   x19,0(sp)    // restore saved x19
    addi sp,sp,8      // return space on stack
    jalr x0,0(x1)     // return
```

How to Load 32-bit Immediates?

- How to implement the C code below in RISC-V?

```
LargeConstant = 3998976;
```

- Intuitive idea: use **addi**

```
addi x19,x0,3998976
```

  - Suppose **LargeConstant** is mapped to register x19
  - But, I-format only allows 12-bit immediate → cannot hold a large constant such as 3998976=0x003D0500
- One solution: use **addi** + **slli** (only need 32 bits)

```
addi x19,x0,0          // 0=0x00
slli x19,x19,12
addi x19,x0,976        // 976=0x3D0
slli x19,x19,12
addi x19,x0,1280       // 1280=0x500
```

Too slow!

How to Load 32-bit Immediates?

- Better solution: Load Upper Immediate (**`lui`**) + **`addi`**
  - **`lui`** handles left 20 bits of the constant, i.e. 0x003D0
  - **`addi`** handles the right 12 bits of the constant, i.e. 0x500
- Load Upper Immediate (lui)

```
lui    rd,constant                    // U-type
```

Immediate[31-12]	rd	opcode
20 bits	5 bits	7 bits

  - 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             // 976=0x003D0
```

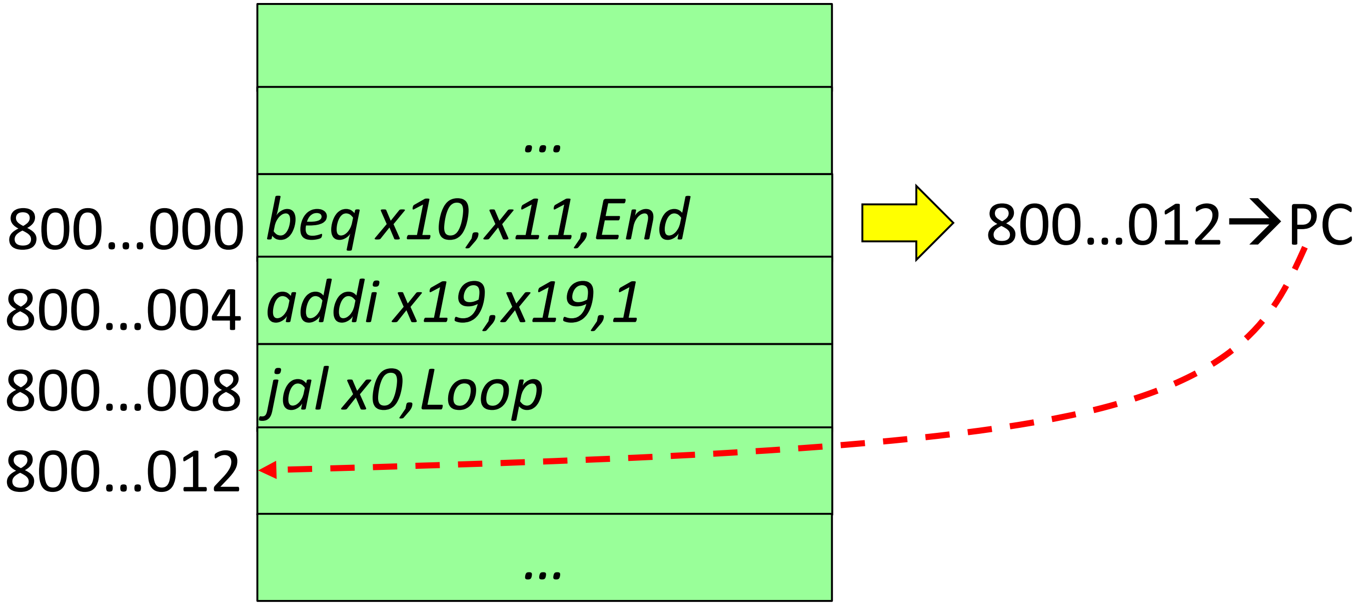
x19	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

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

x19	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

How to Specify Branch Target?

```
Loop:   beq    x10,x11,End
        addi   x19,x19,1
        jal    x0,Loop
End:
```



	…
800…000	*beq x10,x11,End*
800…004	*addi x19,x19,1*
800…008	*jal x0,Loop*
800…012	
	…

800…012→PC

End = 800…012

How to Specify Branch Target?

- Branch instruction format (SB-type)

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

`beq x10,x11,2000` // 2000 = 0 0111 1101 0000

0	111110	01011	01010	000	1000	0	1100011

- But immediate field can only take 12 bits for target address
- How to get full address (64 bits), e.g. 800…012, for the target instruction?

How to Specify Branch Target?

- Observation:
  most branch targets are near, forward or backward, from current instruction (a branch)

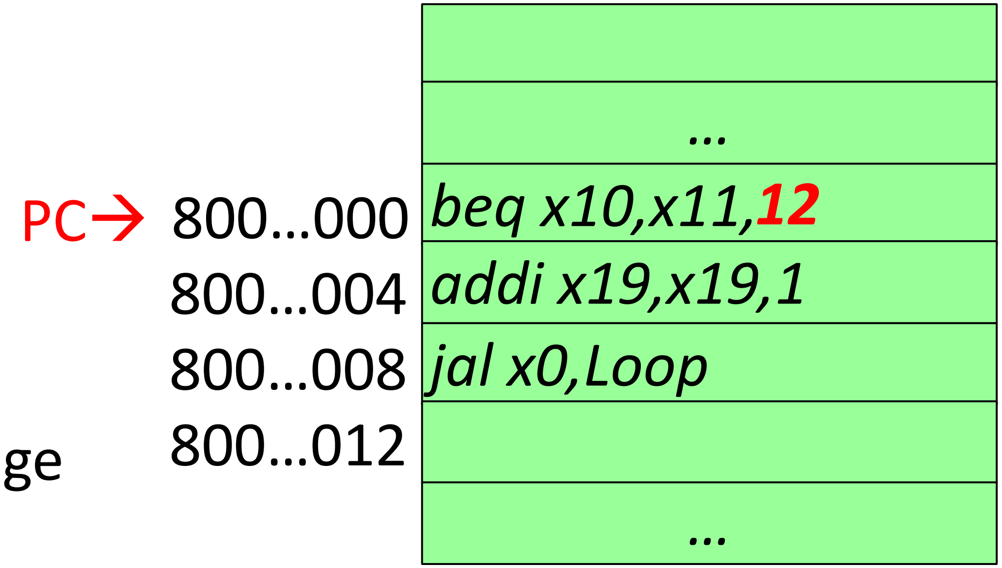
- Solution: PC-relative addressing
  - Use PC to give the 64-bit address and +/- immediate, because most branch targets are near the branch instruction, whose address is currently held in PC
  - Target address = PC + sign-ext(imm12)
    - 12-bit immediate is a signed two’s complement integer, sign-extended to 64 bits, to be added to the PC if branch taken

How to Specify Branch Target?

- So, for our example
  - We should put 12 in the immediate field:

    beq x10,x11,12
  - But, this means we can only branch within a range of +/-2^{11} bytes
  - Since all instructions are 32 bits, why don't we take the immediate value as number of instructions instead of bytes? i.e. within a range of +/-2^{11} words (instructions)
  - So, we can put 3 (words) in the immediate field instead:

    beq x10,x11,3
  - Target address = PC + imm12×4 = PC + {imm12 | 00}



		…
PC→	800…000	*beq x10,x11,12*
	800…004	*addi x19,x19,1*
	800…008	*jal x0,Loop*
	800…012	
		…

How to Specify Branch Target?

- Unfortunately, RISC-V has an additional complication
  - It has a C extension that supports 16-bit instruction encodings in order to reduce code size, e.g.

15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src≠0	C2
C.ADD	dest≠0	src≠0	C2

  - Useful for embedded applications
  - The 16-bit instruction: **`c.add x10,x11`** can be expanded into: **`add x10,x10,x11`**
  - In addition, 16-bit compressed instructions can be mixed with 32-bit instructions

2-address format (page 14, L2-1)

How to Specify Branch Target?

- Now, immediate value in **beq** cannot refer to a word (or a 32-bit instruction); otherwise

  PC = PC + imm12×4

  = 800…000 + 3×4

  = 800…012

  - In fact, no matter whether you put a 2 or 3 in imm23, you cannot jump to 800…010

	…
800…000	*beq x10,x11,3*
800…002	*c.add x19,x20*
800…006	*jal x0,Loop*
800…010	
	…

16-bit ISA "design-in" in RISC-V

- Thus, Immediate value in **beq** must refer to a *halfword* (16-bit instructions) instead, i.e.

  target address = PC + imm12 × **2** = PC + {imm12 | 0}

  - Keep flexibility of supporting 2-byte instructions in RISC-V

How to Specify Branch Target?

- PC-relative addressing
  - Use PC to give the 64-bit address and +/- immediate
    - 12-bit immediate is a signed two's complement integer to be added to the PC if branch taken
  - The addresses actually point to halfwords, i.e.,

    target address = PC + immediate × **2** = PC + {imm | 0}

    - Keep the flexibility of supporting 2-byte instructions in RISC-V, so the branch immediates represent the number of halfwords between the branch instruction and the branch target

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

imm[12] imm[11]

`beq x10,x11,2000` // 2000 = 0 0111 1101 0000

0	111110	01011	01010	000	1000	0	1100011

How to Specify Jump Address?

- Jump and link (**`jal`**): UJ-type

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

`jal x0,2000` // 2000 = 0 0000 0000 0111 1101 0000

0	1111101000	0	00000000	00000	1101111

  - rd ← PC + 4; PC ← PC + {imm | 0} (PC-relative)

- Long jumps to anywhere, i.e., to 32-bit absolute addr.
  - **`lui`**: load address[31:12] to a temp register, e.g. x19
  - **`jalr`**: add address[11:0] to temp reg. and jump to target

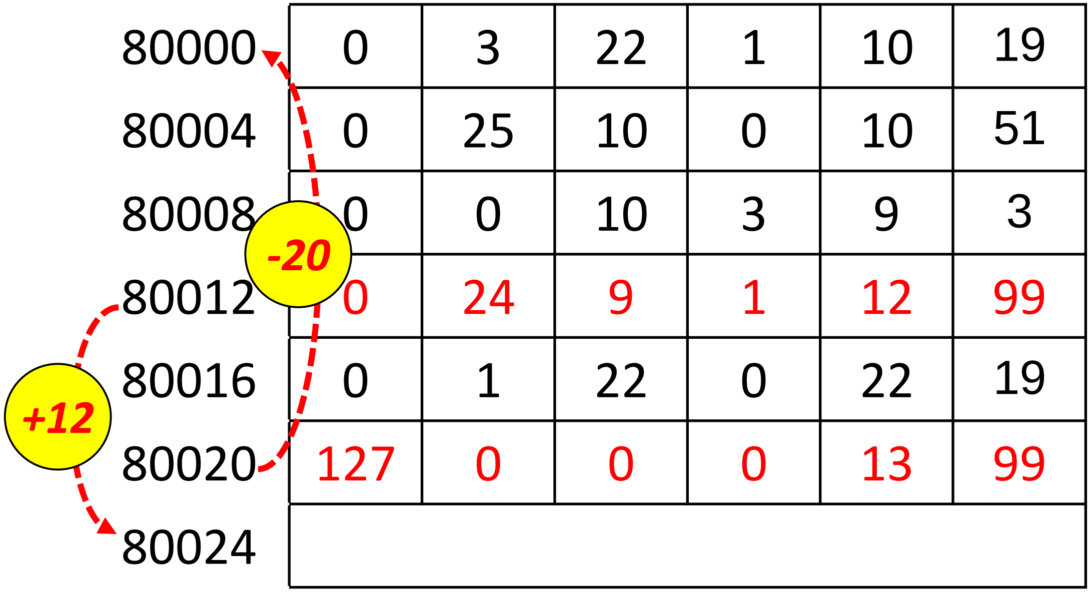
    `jalr x0,100(x19)`

Relative addressing for relocatable code

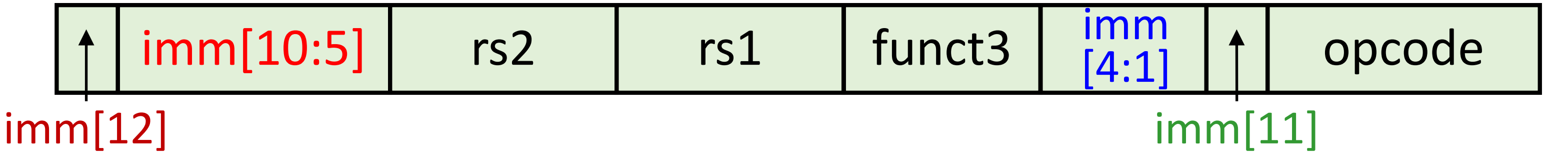
Target Addressing Calculation

- Assume Loop at location 80000

```
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: …
```



80000	0	3	22	1	10	19
80004	0	25	10	0	10	51
80008	0	0	10	3	9	3
80012	0	24	9	1	12	99
80016	0	1	22	0	22	19
80020	127	0	0	0	13	99
80024						

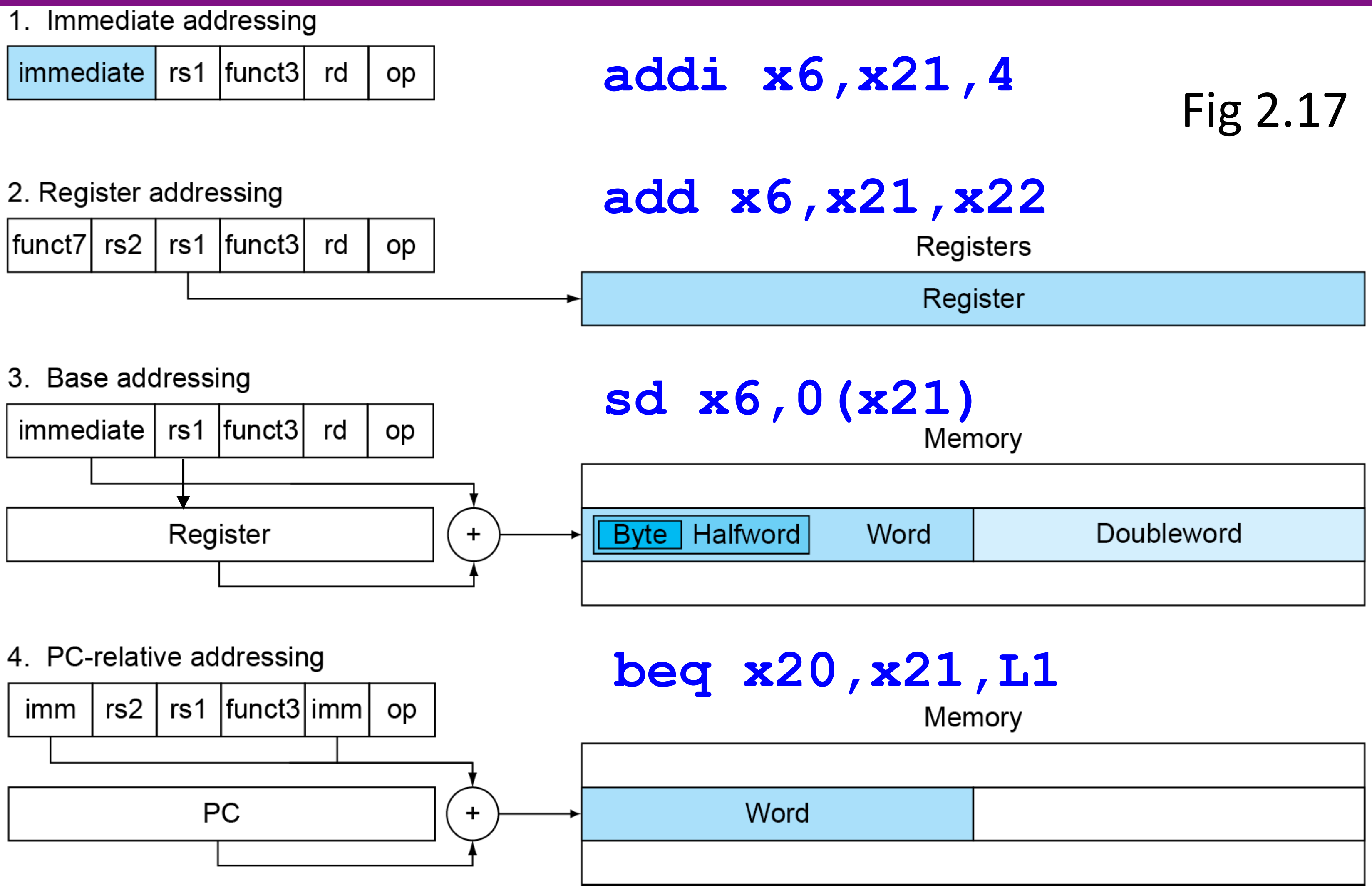


	imm[10:5]	rs2	rs1	funct3	imm [4:1]		opcode

80012: 0000000 11000 01001 001 01100 1100111 → 0 0 000000 0110 0 = +12

80020: 1111111 00000 00000 000 01101 1100111 → 1 1 111111 0110 0 = -20

RISC-V Addressing Modes



```
addi x6,x21,4
```

Fig 2.17

```
add x6,x21,x22
```

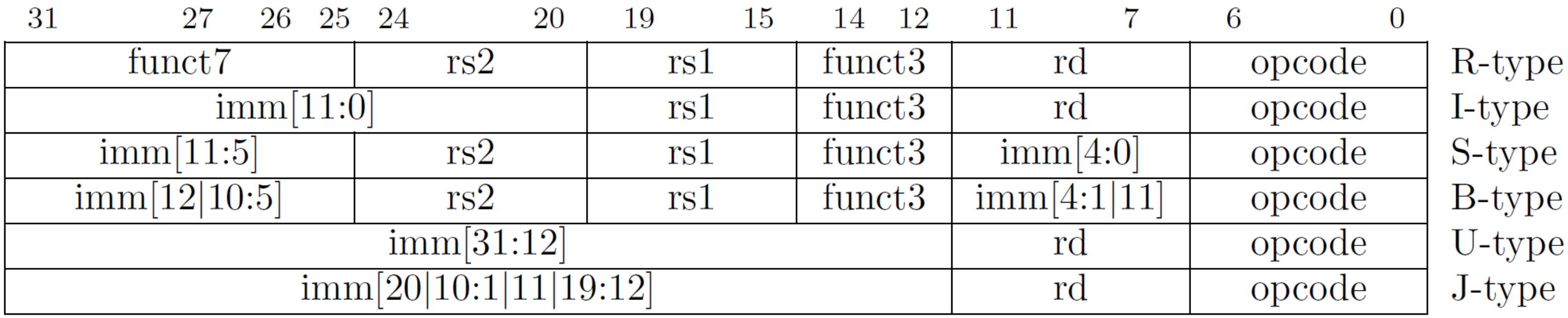
```
sd x6,0(x21)
```

```
beq x20,x21,L1
```

RISC-V Instruction Formats

Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
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

Fig 2.19



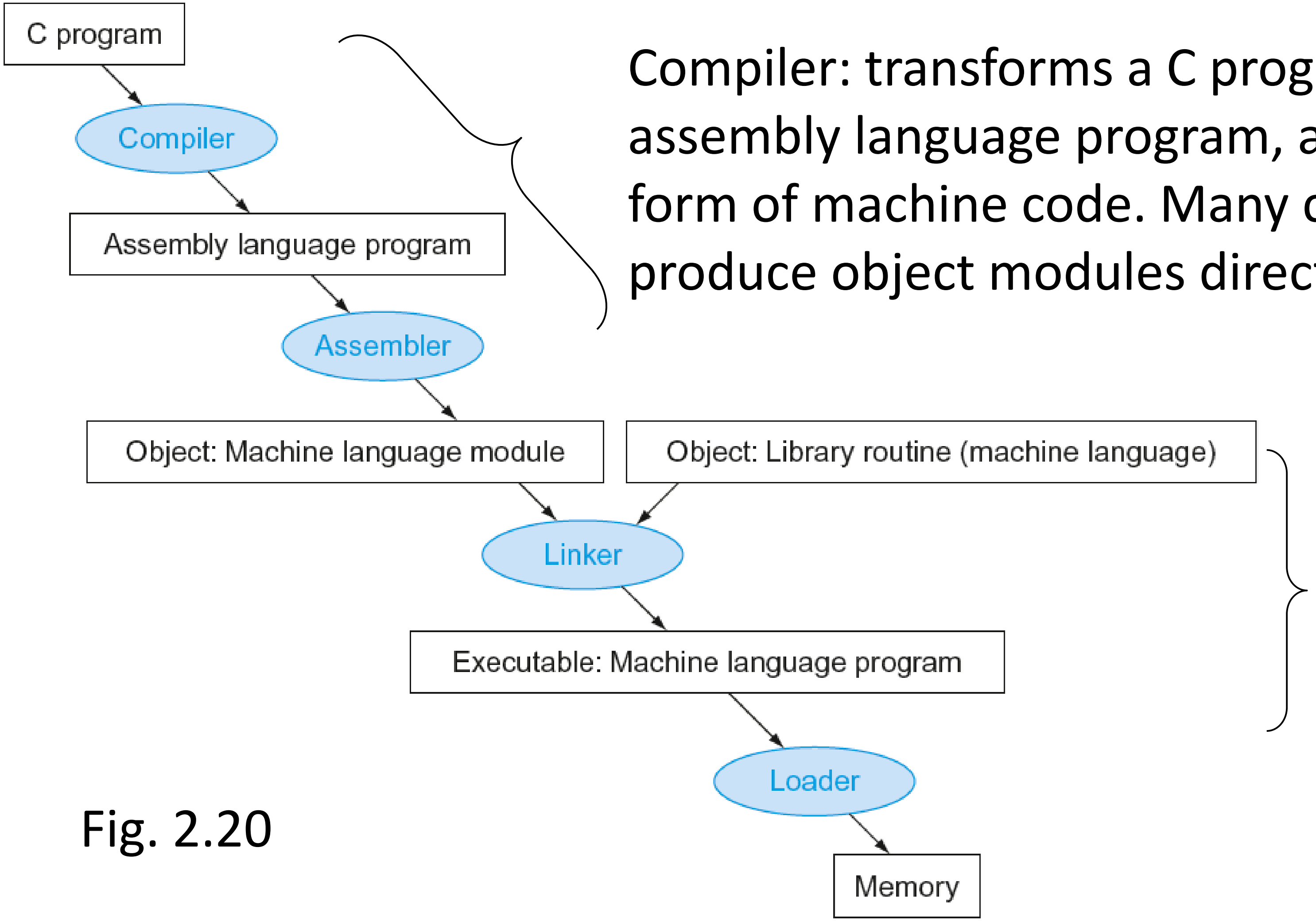
31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Compiler: transforms a C program into an assembly language program, a symbolic form of machine code. Many compilers produce object modules directly

Static linking

Fig. 2.20

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm[20]<<12) + PC
    - add upper immediate to pc: load 20-bit immed into bits [31:12] of rd, sign-extend to bits [63:32], clear bits [11:0], add PC to rd
    - Followed by jalr (adds 12-bit immed) for long jump
      `jalr x0,100(x19)`
  - 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-bit wide, 32-bit operations

Instruction Set Extensions

- I: base architecture (51 instructions)
- M: integer multiply, divide, remainder (13 instr.)
- A: atomic memory operations (22 instr.)
- F: single-precision floating point (30 instr.)
- D: double-precision floating point (32 instr.)
- C: compressed instructions (36 instr.)
  - 16-bit encoding for frequently used instructions

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Fallacies and Pitfalls

- 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 instr.
- Use assembly code for high performance
  - Modern compilers are better at dealing with modern proc.
  - More lines of code ⇒ more errors and less productivity
- Backward compatibility ⇒ instruction set doesn't change
  - But they do increase more instructions

Pitfalls

- Sequential word or doubleword addresses in machines with byte addressing are not differ by 1
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable, e.g., an local array, after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
  - c.f. x86

Supplementary Slides

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Assembler: Producing an Object Module

- Assembler:
  - Translates program into machine instructions, determines addresses of labels by *symbol table*, generates *object files*
- Object file:
  - Including code and information for building a complete program from pieces and placing code properly in memory
  - Header: size and positions of various pieces of information
  - Text segment: translated machine instructions
  - Static data segment: data allocated for the life of program
  - Relocation info: for instructions and data words that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Assembler Pseudoinstructions

- Most assembly instructions represent machine instructions one-to-one
- Pseudoinstructions: assembly instructions defined by the assembler to help assembly programming, but they are not really implemented by the hardware, because they can be realized by true instructions

```
li  x9,123      ➔   addi x9,x0,123
j   L1          ➔   jal  x0,L1
mv  x10,x11     ➔   addi x10,x11,0
and x9,x10,15   ➔   andi x9,x10,15
```

Linker: Linking Object Modules

- Produces an executable image and stores it in a file

1. Merges object modules by placing code and data modules symbolically in memory
2. Determine addresses of data and instruction labels using relocation information and symbol table
3. Patch internal and external references
4. Determine memory locations that each module will occupy
    - All absolute references must be relocated to reflect true loc.

Example in page 128 of textbook

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

Loader: Loading a Program

Load from executable 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 for **`main()`** 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

- Problems with static libraries
  - Library routines become part of the executable code → cannot use newer version libraries unless rebuilt
  - It loads all routines in the library that are called in the executable, even if those calls are not executed
- Dynamically linked libraries (DLLs): only link/load library procedures 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

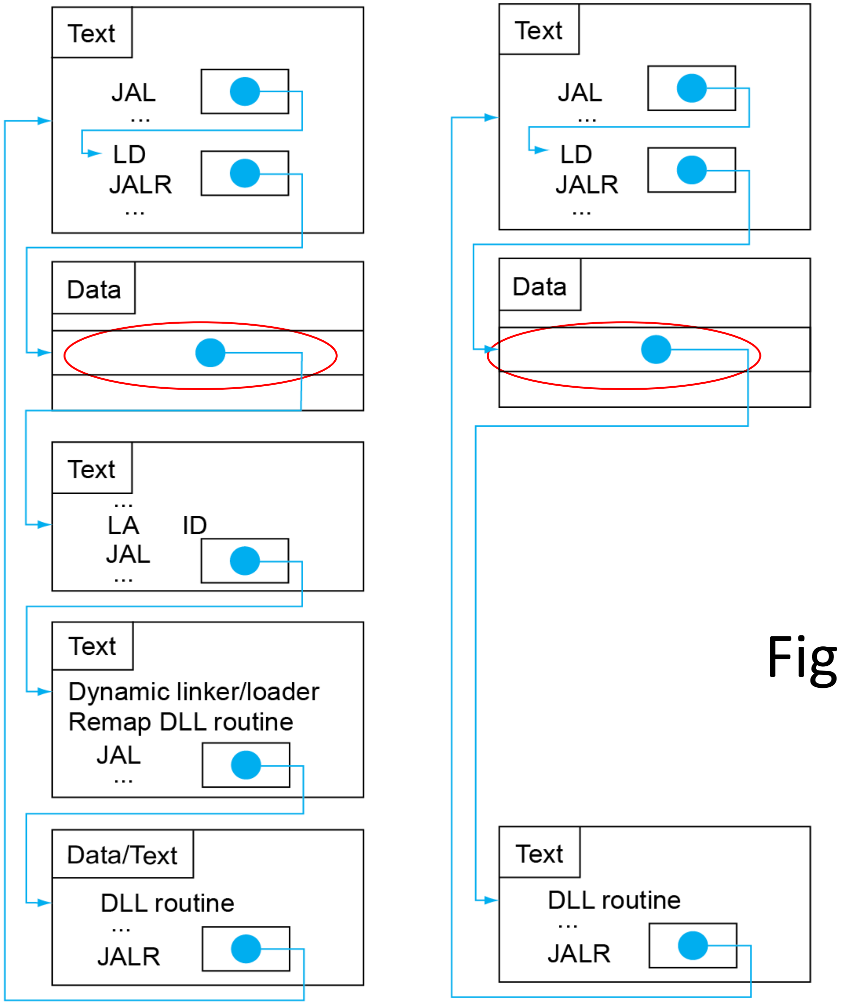
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

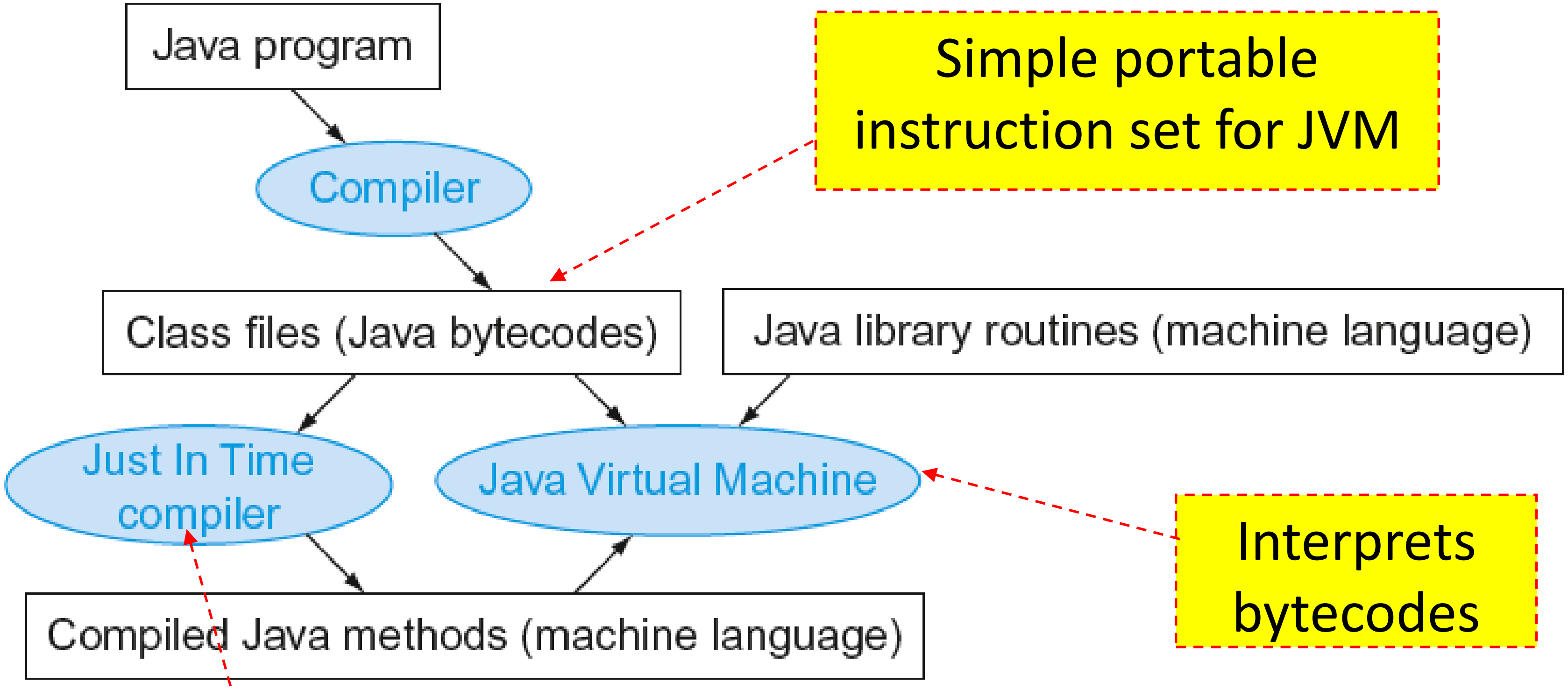


(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Fig. 2.21

Java: Different Way of Translating/Startup


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

Simple portable instruction set for JVM

Interprets bytecodes

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

Fig. 2.23

▷

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

C Bubble Sort Example: Swap Procedure

```
void swap(long long int v[], size_t 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**

```
swap: slli x6,x11,3   // x6 = k * 8
      add  x6,x10,x6 // x6=v+(k*8) addr of v[k]
      ld   x5,0(x6)   // x5 (temp) = v[k]
      ld   x7,8(x6)   // x7 = v[k+1]
      sd   x7,0(x6)   // v[k] = x7 (v[k+1])
      sd   x5,8(x6)   // v[k+1] = x5 (temp)
      jalr x0,0(x1)   // return to caller
```

C Sort Example: Sort Procedure

- Non-leaf (calls swap):

```
void sort(long long 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) {
```

```
   addi x19,x0,0   // x19 = i = 0
for1tst:
   bge  x19,x11,exit1  // exit1 if (i≥n)
      {...}        // body of "for i"
   addi x19,x19,1  // i += 1
   beq  x0,x0,for1tst // branch to outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:

```
  for (j=i-1; j>=0 && v[j]>v[j+1]; j-=1) {
```

```
      addi x20,x19,-1   // x20 = j = i - 1
for2tst:
      blt  x20,x0,exit2// if(j < 0), exit loop
      slli x5,x20,3     // reg x5 = j * 8
      add  x5,x10,x5    // reg x5 = v[ ] + (j * 8)
      ld   x6,0(x5)     // reg x6 = v[j]
      ld   x7,8(x5)     // reg x7 = v[j + 1]
      ble  x6,x7,exit2 // v[j] =< v[j+1], no swap
      jal  x1,swap      // call swap
      addi x20,x20,-1   // j -= 1
      beq  x0,x0,for2tst // check next array element
exit2:
```

Passing Parameters

- Both sort() and swap() need to use arguments stored in x10 and x11 throughout the execution
  - Solution: copy x10, x11 in sort() to saved registers, x21, x22

```
                add  x21,x10,x0   // copy parameter x10 into x21
                add  x22,x11,x0   // copy parameter x11 into x22
                addi x19,x0,0     // i = 0
for1tst:        bge  x19,x22,exit1
                addi x20,x19,-1   // x20 = j = i - 1
for2tst:        blt  x20,x0,exit2// if(j < 0), exit loop
                slli x5,x20,3     // reg x5 = j * 8
                add  x5,x21,x5    // reg x5 = v[ ] + (j * 8)
                ld   x6,0(x5)     // reg x6 = v[j]
                ld   x7,8(x5)     // reg x7 = v[j + 1]
                ble  x6,x7,exit2 // v[j] =< v[j+1], no swap
                add  x10,x21,x0   // prepare 1st swap parameter
                add  x11,x20,x0   // prepare 2nd swap parameter
                jal  x1,swap      // call swap
                addi x20,x20,-1   // j -= 1
                beq  x0,x0,for2tst // check next array element
```

Preserving Registers

- Preserve saved registers:

```
        addi sp,sp,-40   // make room on stack
        sd   x1,32(sp)   // save x1 on stack
        sd   x22,24(sp)  // save x22 on stack
        sd   x21,16(sp)  // save x21 on stack
        sd   x20,8(sp)   // save x20 on stack
        sd   x19,0(sp)   // save x19 on stack
```

- Restore saved registers:

```
exit1:  sd   x19,0(sp)   // restore x19 from stack
        sd   x20,8(sp)   // restore x20 from stack
        sd   x21,16(sp)  // restore x21 from stack
        sd   x22,24(sp)  // restore x22 from stack
        sd   x1,32(sp)   // restore x1 from stack
        addi sp,sp, 40   // restore stack pointer
        jalr x0,0(x1)
```

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Addressing C Arrays

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

Can do better by C pointers, for corresponding directly to memory addresses

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

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

```
   li   x5,0        // i = 0
loop1:
   slli x6,x5,3     // x6 = i * 8
   add  x7,x10,x6   // x7 = address
                    // of array[i]
   sd   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=addr of array[0]
   slli x6,x11,3     // x6 = size * 8
   add x7,x10,x6     // x7 = address
                     // of array[size]
loop2:
   sd x0,0(x5)       // Memory[p] = 0
   addi x5,x5,8      // p = p + 8
   bltu x5,x7,loop2 //if(p<&array[size])
                     // go to loop2
```

Comparison of Array vs. Pointer

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

Outline

- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)
- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

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

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 11	10 6	5 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31 20	19 15	14 12	11 7	6 0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

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

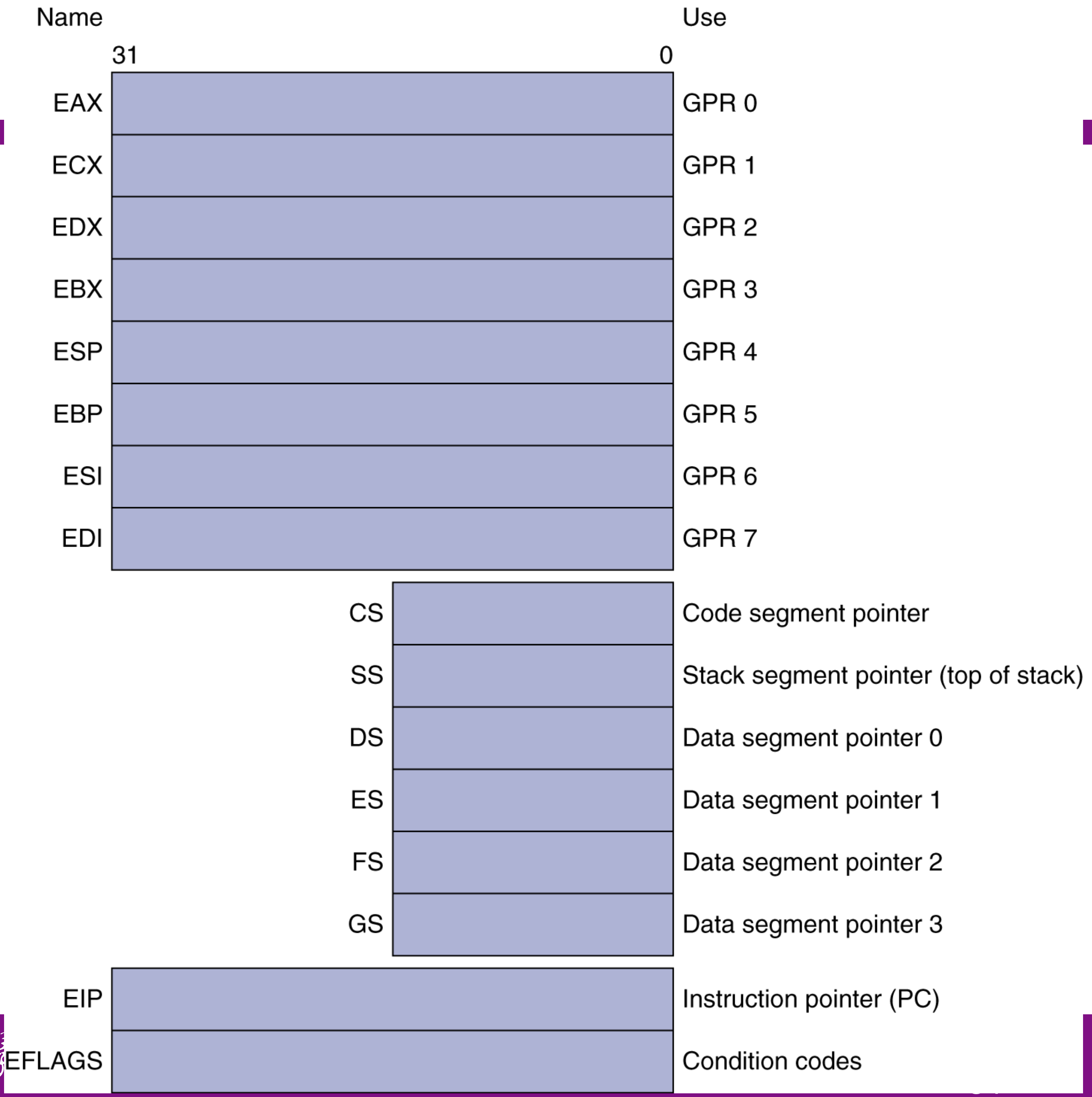
The Intel x86 ISA

- 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) instr.
    - 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 reg.
  - 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



Name	31 … 0	Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
CS		Code segment pointer
SS		Stack segment pointer (top of stack)
DS		Data segment pointer 0
ES		Data segment pointer 1
FS		Data segment pointer 2
GS		Data segment pointer 3
EIP		Instruction pointer (PC)
EFLAGS		Condition codes

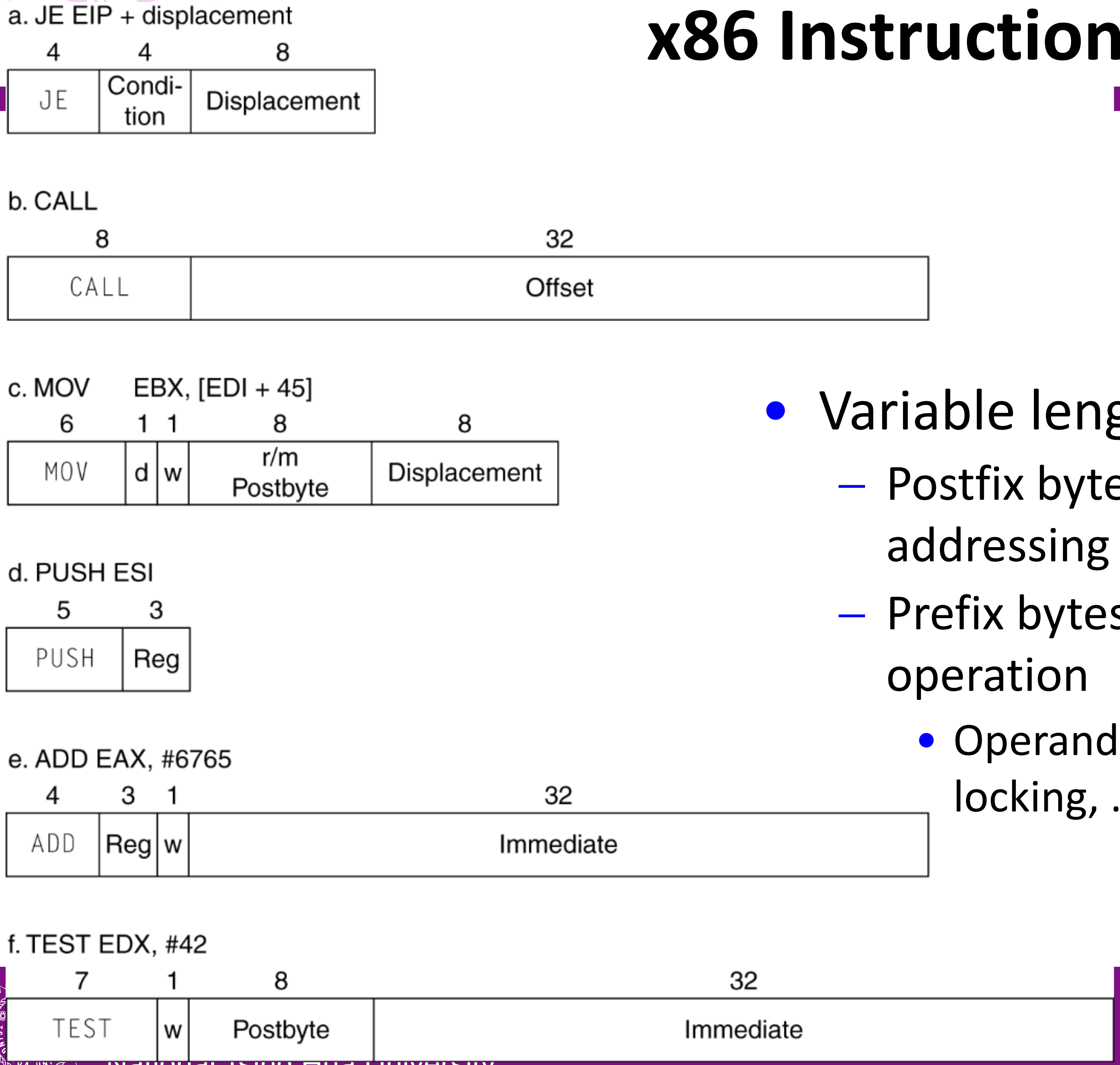
Basic x86 Addressing Modes

- 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

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

- 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
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions