Procedure Calling

§2.8 Supporting Procedures in Computer Hardware

- 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)

Procedure Call Instructions

- Procedure call: jump and link

```
jal x1, ProcedureLabel
```

  - Address of following instruction put in x1
  - Jumps to target address
- Procedure return: jump and link register

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

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

Leaf Procedure Example

- C code:

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

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

Chapter 2 — Instructions: Language of the Computer — 3

Leaf Procedure Example

- RISC-V code:

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

Chapter 2 — Instructions: Language of the Computer — 4

Local Data on the Stack

High address

SP →

Contents of register x5
Contents of register x6
Contents of register x20

Low address

(a) (b) (c)

Register Usage

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

Non-Leaf Procedures

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

Chapter 2 — Instructions: Language of the Computer — 7

Non-Leaf Procedure Example

- C code:

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

  - Argument n in x10
  - Result in x10

Chapter 2 — Instructions: Language of the Computer — 8

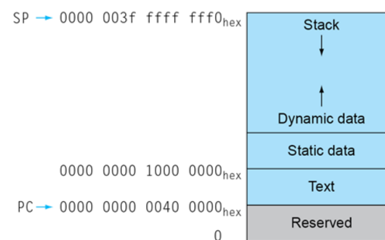
Leaf Procedure Example

- RISC-V code:

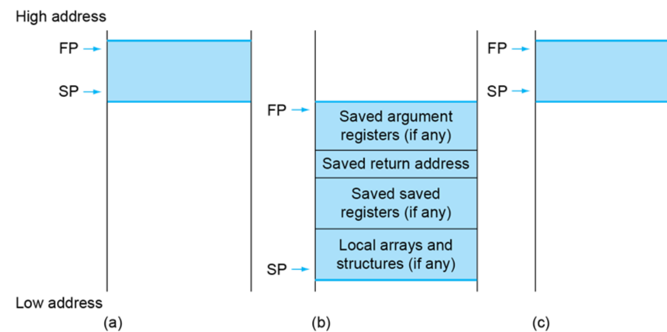
```
fact:
    addi sp,sp,-16         Save return address and n on stack
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1         x5 = n - 1
    bge  x5,x0,L1          if n >= 1, go to L1
    addi x10,x0,1          Else, set return value to 1
    addi sp,sp,16          Pop stack, don't bother restoring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

Memory Layout

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



Local Data on the Stack



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

Chapter 2 — Instructions: Language of the Computer — 11

Character Data

§2.9 Communicating with People

- 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

Chapter 2 — Instructions: Language of the Computer — 12

Byte/Halfword/Word Operations

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

Chapter 2 — Instructions: Language of the Computer — 13

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;
}
```

Chapter 2 — Instructions: Language of the Computer — 14

String Copy Example

- RISC-V code:

```
strcpy:
    addi sp,sp,-8       // adjust stack for 1 doubleword
    sd   x19,0(sp)      // push x19
    add  x19,x0,x0      // i=0
L1: add  x5,x19,x10     // 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: ld   x19,0(sp)      // restore saved x19
    addi sp,sp,8        // pop 1 doubleword from stack
    jalr x0,0(x1)       // and return
```

Chapter 2 — Instructions: Language of the Computer — 15

32-bit Constants

§2.10 RISC-V Addressing for Wide Immediates and Addresses

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

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

`lui x19, 976  // 0x003D0`

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

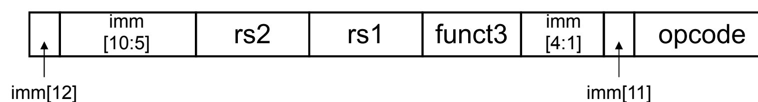
`addi x19,x19,128  // 0x500`

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

Chapter 2 — Instructions: Language of the Computer — 16

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

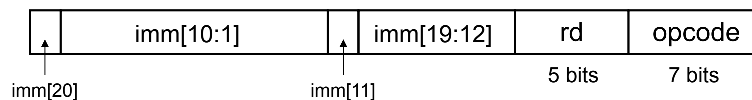


- PC-relative addressing
  - Target address = PC + immediate × 2

Chapter 2 — Instructions: Language of the Computer — 17

Jump Addressing

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

Chapter 2 — Instructions: Language of the Computer — 18

RISC-V Addressing Summary

1. Immediate addressing

immediate	rs1	funct3	rd	op

2. Register addressing

funct7	rs2	rs1	funct3	rd	op

Registers

Register

3. Base addressing

immediate	rs1	funct3	rd	op

Register + → Memory: Byte Halfword Word Doubleword

4. PC-relative addressing

imm	rs2	rs1	funct3	imm	op

PC + → Memory: Word

Chapter 2 — Instructions: Language of the Computer — 19

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

Chapter 2 — Instructions: Language of the Computer — 20

C Sort Example

§2.13 A C Sort Example to Put It All Together

- 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,x11,3    // reg x6 = k * 8
  add  x6,x10,x6   // reg x6 = v + (k * 8)
  ld   x5,0(x6)    // reg x5 (temp) = v[k]
  ld   x7,8(x6)    // reg x7 = v[k + 1]
  sd   x7,0(x6)    // v[k] = reg x7
  sd   x5,8(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 (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

Chapter 2 — Instructions: Language of the Computer — 23

The Outer Loop

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

```
  li   x19,0            // i = 0
for1tst:
  bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

  (body of outer for-loop)

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

Chapter 2 — Instructions: Language of the Computer — 24

The Inner Loop

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

```
    addi x20,x19,-1    // j = i -1
for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    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   // go to exit2 if x6 ≤ x7
    mv   x21, x10      // copy parameter x10 into x21
    mv   x22, x11      // copy parameter x11 into x22
    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
    j    for2tst       // branch to test of inner loop
exit2:
```

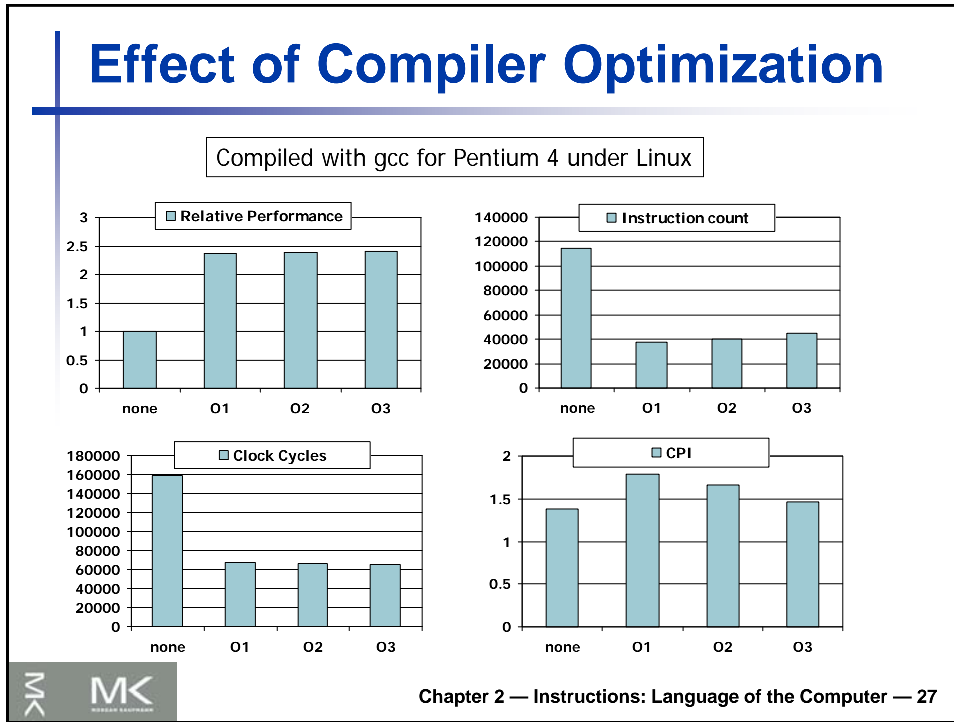
Preserving Registers

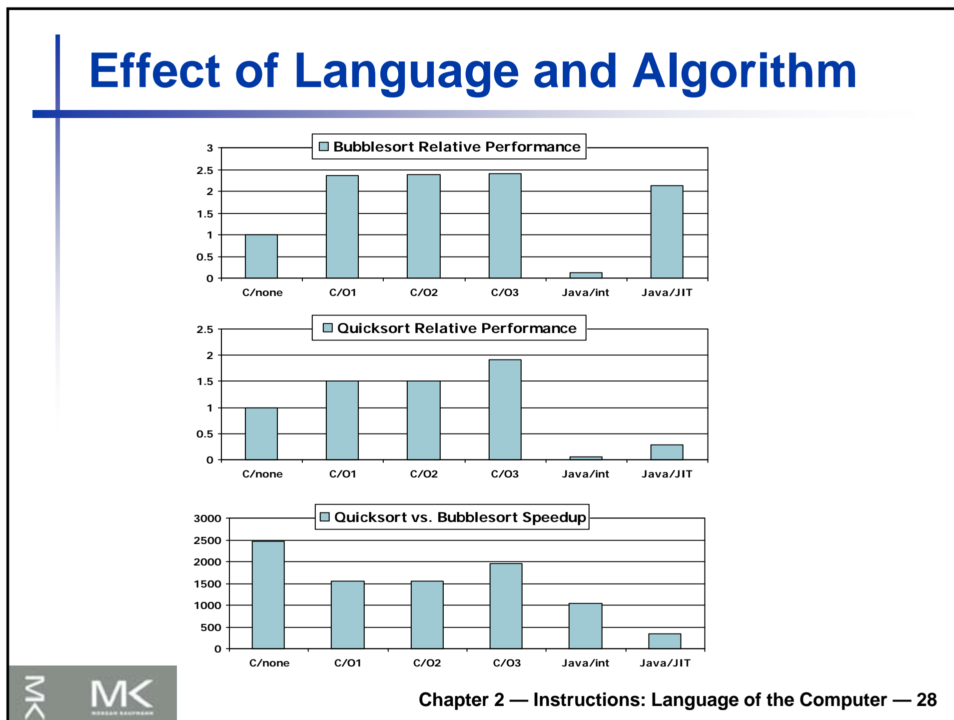
- Preserve saved registers:

```
addi sp,sp,-40  // make room on stack for 5 regs
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)
```


Effect of Compiler Optimization
Compiled with gcc for Pentium 4 under Linux
Relative Performance
3
2.5
2
1.5
1
0.5
0
none
O1
O2
O3
Instruction count
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3
Clock Cycles
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3
CPI
2
1.5
1
0.5
0
none
O1
O2
O3
Chapter 2 — Instructions: Language of the Computer — 27


Effect of Language and Algorithm
Bubblesort Relative Performance
3
2.5
2
1.5
1
0.5
0
C/none
C/O1
C/O2
C/O3
Java/int
Java/JIT
Quicksort Relative Performance
2.5
2
1.5
1
0.5
0
C/none
C/O1
C/O2
C/O3
Java/int
Java/JIT
Quicksort vs. Bubblesort Speedup
3000
2500
2000
1500
1000
500
0
C/none
C/O1
C/O2
C/O3
Java/int
Java/JIT
Chapter 2 — Instructions: Language of the Computer — 28

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!

Chapter 2 — Instructions: Language of the Computer — 29

Arrays vs. Pointers

§2.14 Arrays versus Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Chapter 2 — Instructions: Language of the Computer — 30

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,x5,3 // 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 = address
	               // 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

Chapter 2 — Instructions: Language of the Computer — 31

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

Chapter 2 — Instructions: Language of the Computer — 32