

Chapter 2

Procedure Call and Other Architectures

**Outline**

■ Supporting Procedures in Computer Hardware ■ Communicating with People

■ RISC-V Addressing for Wide Immediates and Addresses

■ Parallelism and Instructions: Synchronization ■ Translating and Starting a program

■ A C Sort Example to Put it All Together

■ Arrays versus Pointers

■ Real Stuff: MIPS Instructions

■ Real Stuff: x86 Instructions

■ Real Stuff: The Rest of the RISC-V Instruction Set ■ Fallacies and Pitfalls

■ Concluding Remarks

**Chapter 2 — Instructions: Language of the Computer — 2**

§

**Procedure Call: An Example**

2

.

8

S

u

p

p

o

#include <stdio.h>

r

t

i

n

#include <stdint.h>

g

P

int64\_t sum\_array(int64\_t nums[], **int** size){

r

o

int64\_t sum=0;

c

e

d

**for** (**int** i = 0; i < size; ++i){

u

r

sum=sum+nums[i];

e

s

}

i

n

C

**return** sum;

o

m

}

p

u

t

e

**int main**(){

r

H

**int** size = 4;

a

r

d

int64\_t list[] = {3, 5, 4, 6};

w

a

int64\_t output = 0;

r

e

output=sum\_array(list, size);

**printf**("Sum=%lld\n", output); **return** 0;

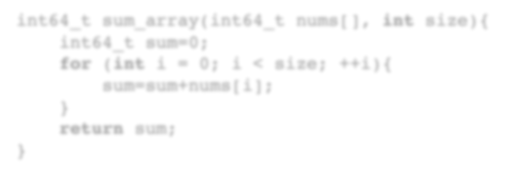
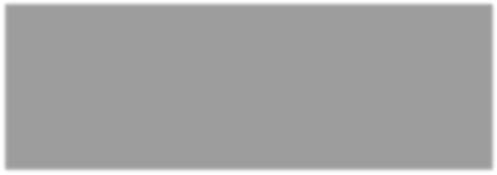
int64\_t == long long int (8 bytes)

**Chapter 2 — Instructions: Language of the Computer — 3**

}

**Example in Memory**

Assume we compile the program and put the binary codes in memory

int64\_t sum\_array(int64\_t nums[], **int** size){ int64\_t sum=0; 

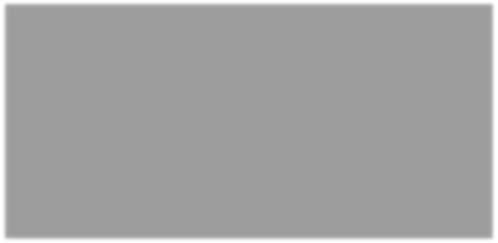
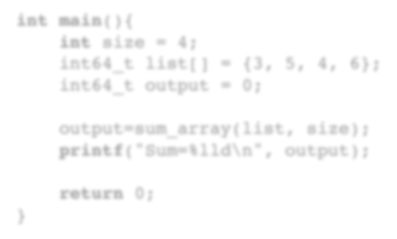
**for** (**int** i = 0; i < size; ++i){

sum=sum+nums[i];

}

**return** sum;

}

**int main**(){ 

**int** size = 4;

int64\_t list[] = {3, 5, 4, 6};

int64\_t output = 0;

output=sum\_array(list, size);

Memory space

**printf**("Sum=%lld\n", output);

**return** 0;

} **4**

§

**Procedure Calling**

2

.

8

S

u

p

p

o

■ Steps required

r

t

i

n

g

1. Place parameters in registers x10 to x17

P

r

o

c

e

2. Transfer control to procedure

d

u

r

e

3. Acquire storage for procedure

s

i

n

C

4. Perform procedure’s operations

o

m

p

5. Place result in register for caller

u

t

e

r

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

H

a

r

d

w

a

r

e

**Chapter 2 — Instructions: Language of the Computer — 5**

**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)

■ Jumps to 0 + address in x1

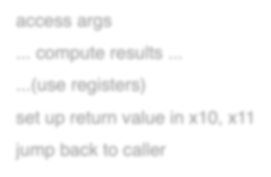
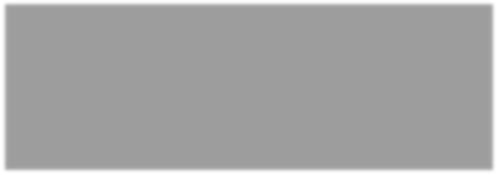
■ Use x0 as rd

■ We do not need the return address and x0 cannot be changed (always 0)

■ Can also be used for computed jumps ■ e.g., for case/switch statements

**Chapter 2 — Instructions: Language of the Computer — 6**

**Procedure Call Assembly Steps**

Memory 

space

**call**

**return**

access args

... compute results ...

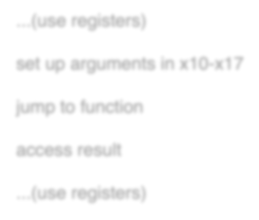
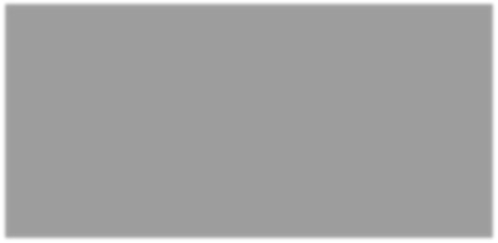
...(use registers)

set up return value in x10, x11 jump back to caller

...(use registers)

set up arguments in x10-x17

**CalleeCaller**

jump to function 

access result

...(use registers)

**7**

**Execution of a Procedure**

1. Place parameters in a place where the procedure can access them (via x10 – x17)

■ If more than 8 parameters, push them into the beginning of stack frame. More later.

2. Transfer control to the procedure

by jal x1, ProcedureAddress

3. Acquire the storage resources needed for the procedure 4. Perform the desired task

5. Place the result value in a place where the calling program can access it (via x10, x11)

6. Return control to the point of origin by jalr x0, 0(x1)

**Chapter 2 — Instructions: Language of the Computer — 8**

**Procedure Call with jal**

ProcedureAddr

Memory 

space

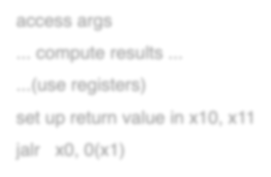
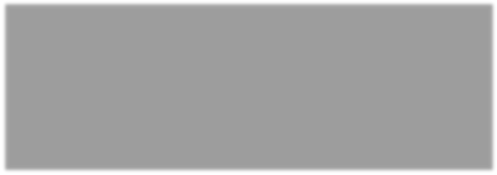
**return**

PC

PC+4

PC+8

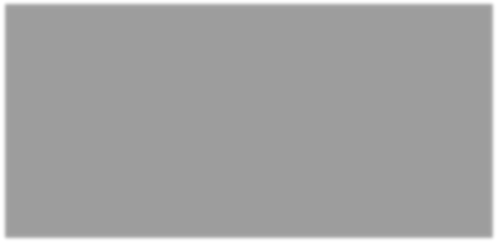
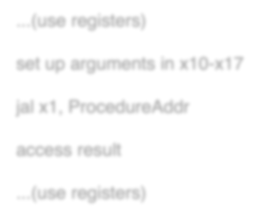
:

access args 

... compute results ...

...(use registers)

set up return value in x10, x11 jalr x0, 0(x1)

...(use registers) 

set up arguments in x10-x17 jal x1, ProcedureAddr

access result

...(use registers)

**Callee**

**Caller**

x1 🡨 PC+4

**call 9**

**Leaf Procedure Example**

■ C code:

int64\_t leaf\_example (int64\_t g, int64\_t h, int64\_t i, int64\_t j) {

int64\_t f;

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

**return** f;

}

■ Arguments g, …, j in x10, …, x13

■ f in x20

■ temporaries x5, x6

■ Need to save x20 on stack (x20 is a saved register)

**Chapter 2 — Instructions: Language of the Computer — 10**

**RISC-V code of Example**

leaf\_example:

addi sp,sp,-8 //Save x20 on stack

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) //Restore x20 from stack

addi sp,sp,8

jalr x0,0(x1) //Return to caller

**Chapter 2 — Instructions: Language of the Computer — 11**

**Save Registers on Stack (Push)**

High address SP 🡪

Low address

SP 🡪

|  |
| --- |

SP 🡪

| value of x20 |
| --- |

addi sp,sp,-8

sd x20,0(sp)

**Chapter 2 — Instructions: Language of the Computer — 12**

**Restore Registers on Stack (Pop)**

SP 🡪

| value of x20 |
| --- |

SP 🡪

SP 🡪

|  |
| --- |

ld x20,0(sp)

addi sp,sp,8

**Chapter 2 — Instructions: Language of the Computer — 13**

**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 — 14**

**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

**Chapter 2 — Instructions: Language of the Computer — 15**

**RISC-V Register Names**

**Register Assembly 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 − x5-7 t0-2 Temporaries Caller x8 s0/fp Saved register/frame pointer Callee x9 s1 Saved register Callee x10-11 a0-1 Function arguments/return values Caller x12-17 a2-7 Function arguments Caller x18-27 s2-11 Saved registers Callee x28-31 t3-6 Temporaries Caller

**Chapter 2 — Instructions: Language of the Computer — 16**

**Non-Leaf Procedure Example**

■ C code:

int64\_t fact (int64\_t n){

**if** (n < 1) **return** 1;

**else return** n \* fact(n - 1); }

■ Argument n in x10

■ Result in x10

**Chapter 2 — Instructions: Language of the Computer — 17**

**RISC-V code of Example**

fact:

addi sp,sp,-16

sd x1,8(sp) //Save return address and n on stack sd x10,0(sp) //x10 = n

addi x5,x10,-1 //x5 = n-1

bge x5,x0,L1 //if (x5>0), go to L1

addi x10,x0,1 //else return x10 = 1

addi sp,sp,16 //pop back and discard all stack values jalr x0,0(x1) //return to (x1)

L1: addi x10,x10,-1//n = n-1

jal x1,fact //call fact(n-1)

addi x6,x10,0 //move return of fact(n-1) in x10 to x6 ld x10,0(sp) //restore caller's n to x10

ld x1,8(sp) //restore caller's return address to x1 addi sp,sp,16 //pop back stack address

mul x10,x10,x6//return n \* fact(n-1)

jalr x0,0(x1) //return

**Chapter 2 — Instructions: Language of the Computer — 18**

**Issue 1: Register Conflict**

■ Caller and callee both use the same registers

■ One example solution

■ Caller saves temporary registers (x5-x7, x28- x31) into a memory stack if they are to be used later

■ Callee saves saved registers (x8-x9, x18-x27) if it uses them

■ Both share the task of saving to memory

**Chapter 2 — Instructions: Language of the Computer — 19**

**Issue 2: Need More Arguments**

■ Caller need to pass more arguments than 8 (x10, ..., x17) to callee

■ Solution

■ Place extra variables and extra arguments onto stack (a LIFO queue)

■ x2 (stack pointer) points to most recently allocated address

**Chapter 2 — Instructions: Language of the Computer — 20**

**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 ■ May use x8 for FP

**Chapter 2 — Instructions: Language of the Computer — 21**

**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

x3

**Chapter 2 — Instructions: Language of the Computer — 22**

§

**Character Data**

2

.

9

C

o

m

m

■ Byte-encoded character sets

u

n

i

c

a

■ ASCII: 128 characters

t

i

n

g

■ 95 graphic, 33 control

w

i

t

h

■ Latin-1: 256 characters

P

e

o

p

■ ASCII, +96 more graphic characters

l

e

■ 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 — 23**

**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 — 24**

**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 — 25**

**RISC-V code of strcpy Example**

**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 — 26**

§

**32-bit Constants**

2

.

1

0

R

I

S

C

-

■ Most constants are small

V

A

d

d

■ 12-bit immediate is sufficient

r

e

s

s

■ For the occasional 32-bit constant

i

n

g

f

o

lui rd, constant

r

W

i

d

e

■ Copies 20-bit constant to bits [31:12] of rd

I

m

m

■ Extends bit 31 to bits [63:32]

e

d

i

a

■ Clears bits [11:0] of rd to 0

t

e

s

a

n

lui x19, 976 // 0x003D0

d

A

d

0000 0000 0000 0000 0000 0000 0011 1101 0000

0000 0000 0000 0000 0000 0000 0000

d

r

e

s

addi x19,x19,128 // 0x500

s

e

s

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

**Chapter 2 — Instructions: Language of the Computer — 27**

**Branch Addressing**

■ Branch instructions specify

■ Opcode, two registers, target address ■ Most branch targets are near branch ■ Forward or backward

■ SB format:

rs2 rs1 funct3 opcode imm imm

[10:5]

[4:1]

imm[12] imm[11]■ PC-relative addressing

■ Target address = PC + immediate × 2

**Chapter 2 — Instructions: Language of the Computer — 28**

**Jump Addressing**

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

■ UJ format:

imm[10:1] imm[19:12]

imm[20] imm[11]

rd opcode 5 bits 7 bits

■ Target address = PC + immediate × 2 ■ 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 — 29**

**RISC-V Addressing Summary Chapter 2 — Instructions: Language of the Computer — 30**

**Table of RISC-V Instructions **31

32

**RISC-V Standard Base ISA**

**Encoding**

31 25 24 20 19 15 14 12 11 7 6 0funct7 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 SB-type imm[31:12] rd opcode U-type imm[20, 10:1, 11, 19:12] rd opcode UJ-type

● R-type: arithmetic instructions

● I-type: Loads & immediate arithmetic

● S-type: Stores

● SB-type: Conditional branch format

● UJ-type: Unconditional jump format

● U-type: Upper immediate format 33

§

**Synchronization**

2

.

1

1

P

a

r

a

■ Two processors sharing an area of memory

l

l

e

l

i

s

m

■ P1 writes, then P2 reads

a

n

■ Data race if P1 and P2 don’t synchronize

d

I

n

s

■ Result depends of order of accesses

t

r

u

c

■ Hardware support required

t

i

o

n

s

■ Atomic read/write memory operation

:

S

y

■ No other access to the location allowed between the

n

c

h

read and write

r

o

n

i

z

■ Could be a single instruction

a

t

i

o

n

■ E.g., atomic swap of register ↔ memory

■ Or an atomic pair of instructions

**Chapter 2 — Instructions: Language of the Computer — 34**

**Synchronization in RISC-V**

■ Load reserved: lr.d rd,(rs1) ■ Load from address in rs1 to rd

■ Place reservation on memory address

■ Store conditional: sc.d rd,(rs1),rs2 ■ Store from rs2 to address in rs1

■ Succeeds if location not changed since the lr.d ■ Returns 0 in rd

■ Fails if location is changed

■ Returns non-zero value in rd

**Chapter 2 — Instructions: Language of the Computer — 35**

**Synchronization in RISC-V**

■ Example 1: atomic swap (to test/set lock variable) again: lr.d x10,(x20)

sc.d x11,(x20),x23 // X11 = status

bne x11,x0,again // branch if store failed addi x23,x10,0 // X23 = loaded value

■ Example 2: lock

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

■ Unlock:

sd x0,0(x20) // free lock

**Chapter 2 — Instructions: Language of the Computer — 36**

§

**Translation and Startup**

2

.

1

2

T

r

a

n

s

l

a

t

i

n

g

a

n

Many compilers produce

d

object modules directly

S

t

a

r

t

i

n

g

a

P

r

o

g

r

a

m

Static linking

**Chapter 2 — Instructions: Language of the Computer — 37**

**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

**Chapter 2 — Instructions: Language of the Computer — 38**

**Linking Object Modules**

■ Produces an executable image

1. Merges segments

2. Resolve labels (determine their addresses) 3. Patch location-dependent and external refs ■ Could leave location dependencies for fixing by a relocating loader

■ But with virtual memory, no need to do this ■ Program can be loaded into absolute location in virtual memory space

**Chapter 2 — Instructions: Language of the Computer — 39**

**Loading a Program**

■ Load from image file on disk into memory 1. Read header to determine segment sizes 2. Create virtual address space

3. Copy text and initialized data into memory ■ Or set page table entries so they can be faulted in 4. Set up arguments on stack

5. Initialize registers (including sp, fp, gp) 6. Jump to startup routine

■ Copies arguments to x10, … and calls main ■ When main returns, do exit syscall

**Chapter 2 — Instructions: Language of the Computer — 40**

**Dynamic Linking**

■ Only link/load library procedure when it is called

■ Requires procedure code to be relocatable ■ Avoids image bloat caused by static linking of all (transitively) referenced libraries

■ Automatically picks up new library versions

**Chapter 2 — Instructions: Language of the Computer — 41**

**Lazy Linkage**

Indirection table 

Stub: Loads routine ID,

Jump to linker/loader

Linker/loader code

Dynamically

mapped code

**Chapter 2 — Instructions: Language of the Computer — 42**

**Starting Java Applications**

Simple portable 

instruction set for

the JVM

Compiles

bytecodes of “hot” methods into native

code for host machine

Interprets bytecodes

**Chapter 2 — Instructions: Language of the Computer — 43**

§

**C Sort Example**

2

.

1

3

A

C

■ Illustrates use of assembly instructions for

S

o

a C bubble sort function

r

t

E

■ Swap procedure (leaf)

x

a

m

**void** swap(int64\_t v[], int64\_t k)

p

l

e

t

o

{

P

u

t

int64\_t temp;

I

t

A

l

temp = v[k];

l

T

o

g

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

e

t

h

e

v[k+1] = temp;

r

}

■ v in x10, k in x11, temp in x5

**Chapter 2 — Instructions: Language of the Computer — 44**

**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

**Chapter 2 — Instructions: Language of the Computer — 45**

**The Sort Procedure in C**

■ Non-leaf (calls swap)

**void** sort (int64\_t 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 — 46**

**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 — 47**

**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:

**Chapter 2 — Instructions: Language of the Computer — 48**

**Revised Code of 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 x22, x11 // copy parameter x11 into x22

mv x11, x20 // second swap parameter is j (pass j with x11) jal x1,swap // call swap (base address of v[] is in x10) mv x11, x22 // Restore x11 (n)

addi x20,x20,-1 // j –= 1

j for2tst // branch to test of inner loop

exit2:

**Chapter 2 — Instructions: Language of the Computer — 49**

**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:

ld x19,0(sp) // restore x19 from stack

ld x20,8(sp) // restore x20 from stack

ld x21,16(sp) // restore x21 from stack

ld x22,24(sp) // restore x22 from stack

ld x1,32(sp) // restore x1 from stack

addi sp,sp, 40 // restore stack pointer

jalr x0,0(x1)

**Chapter 2 — Instructions: Language of the Computer — 50**

**Effect of Compiler Optimization** Compiled with gcc for Pentium 4 under Linux

**3**

**2.5 2**

**1.5 1**

**0.5 0**

**Relative Performance**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**none O1 O2 O3**

**140000 120000 100000**

**80000 60000 40000 20000 0**

**Instruction count**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

**none O1 O2 O3**

**180000 160000 140000 120000 100000**

**80000 60000 40000 20000 0**

**Clock Cycles**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**none O1 O2 O3**

**2**

**1.5 1**

**0.5 0**

**CPI**

|  |
| --- |
|  |
|  |
|  |

**none O1 O2 O3**

**Chapter 2 — Instructions: Language of the Computer — 51**

**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**

**3000 2500 2000 1500 1000 500**

**0**

**Quicksort vs. Bubblesort Speedup**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**C/none C/O1 C/O2 C/O3 Java/int Java/JIT**

**Chapter 2 — Instructions: Language of the Computer — 52**

**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 — 53**

§

**Arrays vs. Pointers**

2

.

1

4

A

r

r

a

y

■ Array indexing involves

s

v

e

r

■ Multiplying index by element size

s

u

s

P

■ Adding to array base address

o

i

n

t

e

■ Pointers correspond directly to memory

r

s

addresses

■ Can avoid indexing complexity

**Chapter 2 — Instructions: Language of the Computer — 54**

**Example: Clearing an Array**

clear1(int array[], int size) { int i;

for (i = 0; i < size; i += 1) array[i] = 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

clear2(int \*array, int size) {

int \*p;

for (p = &array[0]; p < &array[size]; p = p + 1)

\*p = 0;

}

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 — 55**

**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 — 56**

§

2

**MIPS Instructions**

.

1

6

R

e

a

■ MIPS: commercial predecessor to RISC-V

l

S

t

u

■ Similar basic set of instructions

f

f

:

M

■ 32-bit instructions

I

P

S

■ 32 general purpose registers, register 0 is always 0

I

n

s

■ 32 floating-point registers

t

r

u

c

■ Memory accessed only by load/store instructions

t

i

o

n

■ Consistent use of addressing modes for all data sizes s

■ 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

**Chapter 2 — Instructions: Language of the Computer — 57**

**Instruction Encoding**

**Chapter 2 — Instructions: Language of the Computer — 58**

**ARM & MIPS Similarities**

■ ARM: the most popular embedded core ■ Similar basic set of instructions to MIPS

ARM MIPS

Date announced 1985 1985 Instruction size 32 bits 32 bits Address space 32-bit flat 32-bit flat Data alignment Aligned Aligned Data addressing modes 9 3 Registers 15 × 32-bit 31 × 32-bit

Input/output Memory mapped

Memory mapped

**Chapter 2 — Instructions: Language of the Computer — 59**

**Compare and Branch in ARM**

■ Uses condition codes for result of an arithmetic/logical instruction

■ Negative, zero, carry, overflow

■ Compare instructions to set condition codes without keeping the result

■ Each instruction can be conditional ■ Top 4 bits of instruction word: condition value ■ Can avoid branches over single instructions

**Chapter 2 — Instructions: Language of the Computer — 60**

**Instruction Encoding**

****

**Chapter 2 — Instructions: Language of the Computer — 61**

§

**The Intel x86 ISA**

2

.

1

7

R

e

a

l

■ Evolution with backward compatibility

S

t

u

f

f

■ 8080 (1974): 8-bit microprocessor

:

x

8

■ Accumulator, plus 3 index-register pairs

6

I

n

■ 8086 (1978): 16-bit extension to 8080

s

t

r

u

c

■ Complex instruction set (CISC)

t

i

o

n

■ 8087 (1980): floating-point coprocessor

s

■ 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

**Chapter 2 — Instructions: Language of the Computer — 62**

**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

**Chapter 2 — Instructions: Language of the Computer — 63**

**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

**Chapter 2 — Instructions: Language of the Computer — 64**

**Basic x86 Registers**

****

**Chapter 2 — Instructions: Language of the Computer — 65**

**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 = Rbase + displacement

■ Address = Rbase + 2scale × Rindex (scale = 0, 1, 2, or 3) ■ Address = Rbase + 2scale × Rindex + displacement

**Chapter 2 — Instructions: Language of the Computer — 66**

**x86 Instruction Encoding**

■ Variable length 

encoding

■ Postfix bytes specify

addressing mode

■ Prefix bytes modify

operation

■ Operand length,

repetition, locking, …

**Chapter 2 — Instructions: Language of the Computer — 67**

**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

**Chapter 2 — Instructions: Language of the Computer — 68**

**ARM v8 Instructions**

■ In moving to 64-bit, ARM did a complete overhaul

■ ARM v8 resembles MIPS

■ Changes from v7:

■ No conditional execution field

■ Immediate field is 12-bit constant

■ Dropped load/store multiple

■ PC is no longer a GPR

■ GPR set expanded to 32

■ Addressing modes work for all word sizes

■ Divide instruction

■ Branch if equal/branch if not equal instructions

**Chapter 2 — Instructions: Language of the Computer — 69**

§

**Other RISC-V Instructions**

2

.

1

8

T

h

e

■ Base integer instructions (RV64I)

R

e

s

t

o

■ Those previously described, plus

f

t

h

e

■ auipc rd, immed // rd = (imm<<12) + pc

R

I

S

C

■ follow by jalr (adds 12-bit immed) for long jump

-

V

I

■ slt, sltu, slti, sltui: set less than (like MIPS)

n

s

t

r

u

■ addw, subw, addiw: 32-bit add/sub

c

t

i

o

n

■ sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift

S

e

t

■ 32-bit variant: RV32I

■ registers are 32-bits wide, 32-bit operations

**Chapter 2 — Instructions: Language of the Computer — 70**

**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

**Chapter 2 — Instructions: Language of the Computer — 71**

§

**Fallacies**

2

.

1

9

F

a

l

l

a

■ Powerful instruction ⇒ higher performance

c

i

e

s

■ Fewer instructions required

a

n

d

■ But complex instructions are hard to implement

P

i

t

f

a

■ May slow down all instructions, including simple ones

l

l

s

■ 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

**Chapter 2 — Instructions: Language of the Computer — 72**

**Fallacies**

■ Backward compatibility ⇒ instruction set doesn’t change

■ But they do accrete more instructions x86 instruction set

**Chapter 2 — Instructions: Language of the Computer — 73**

**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

**Chapter 2 — Instructions: Language of the Computer — 74**

§

**Concluding Remarks**

2

.

2

0

C

o

n

■ Design principles

c

l

u

d

i

n

1. Simplicity favors regularity

g

R

e

2. Smaller is faster

m

a

r

3. Good design demands good compromises

k

s

■ Make the common case fast

■ Layers of software/hardware

■ Compiler, assembler, hardware

■ RISC-V: typical of RISC ISAs

■ c.f. x86

**Chapter 2 — Instructions: Language of the Computer — 75**