COEN-2710 Microprocessors - Lecture 2

Instructions: Language of the Computer (Ch.2)

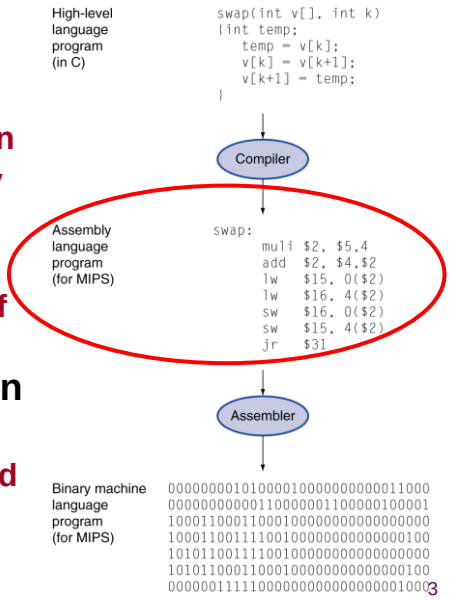
Cris Ababei
Marquette University
Dept. of Electrical and Computer Engineering (ECE)

Goals of this Chapter

- **Instruction Set Architecture (ISA) design**
  1. **Simplicity favors regularity**
  2. **Smaller is faster.**
  3. **Make the common case fast.**
  4. **Good design demands compromise.**
- **Understand the RISC-V Assembly language and be able to write a program using it**

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



3

Outline

- Designing an ISA
- Example ISA: RISC-V
  - Introduction
  - Instruction Formats
  - Writing Assembly
- RISC-V Simulator (used in Project 1)

4

Tasks to Design an ISA

- Decide on:
  - Instructions & Operands
  - Memory model / addressing modes
  - Machine representation
    (binary code for each instruction)
- All design decisions are inter-related
  - Straightforward mapping to hardware implementation
  - Logical register assignments
  - Ease of compilation
- Two differing approaches depending on application
  - CISC - Complex Instruction Set
  - RISC - Reduced Instruction Set

5

5

Typical Instructions (little change over last decades)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input/output (from I/O device) push, pop (to/from stack)
Arithmetic and Logic	Add, Subtract, Multiply, Divide, Not, And, Or, Shift, rotate Integer and/or FP
Control	Jump (unconditional) Branch (conditional) Subroutine calls/returns
Special Operations	Strings, graphics, common ops

6

6

Memory Models: General Purpose Registers

- Several possible memory models
  - Accumulator
  - Stack
  - Load-store via memory
  - Load-store with general purpose registers
- Nearly all machines use general purpose registers
  - Register access is much faster than memory access
    - So memory traffic is reduced, and program is sped-up
  - Registers are easier for a compiler to use
  - Specifying register requires fewer bits than memory address

7

7

Possible Addressing Modes

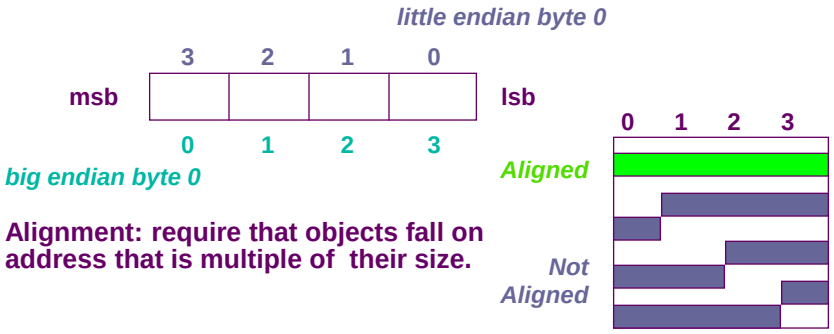
Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

8

8

Byte Arrangement & Alignment

- Big Endian: address of most significant byte = word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Alignment: require that objects fall on address that is multiple of their size.

Outline

- Designing an ISA
- Example ISA: RISC-V
  - Introduction
  - Instruction Formats
  - Writing Assembly
- RISC-V Simulator (used in Project 1)

RISC-V Instruction Set Architecture

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

11

11

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

12

12

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-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
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

Source: **green card** from textbook

Arithmetic Operations

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

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

- All arithmetic operations have this form

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

15

15

Register Operands

- Arithmetic instructions use register operands

- RISC-V has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
    - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a "word"

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

16

16

Register Operand Example

- C code:

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

  - f in x19
  - g in x20
  - h in x21
  - i in x22
  - j in x23
- Compiled RISC-V code:

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

17

17

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *vs.* Big Endian: most-significant byte at least address

18

18

Memory Operand Example

- C code:

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

  - h in x21, base address of A in x22

- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

19

19

Immediate Operands

- Constant data specified in an instruction

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

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

20

20

Conditional Operations

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

21

21

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

22

22

Hexadecimal

- **Base 16**
  - **Compact representation of bit strings**
  - **4 bits per hex digit**

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Example: ECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

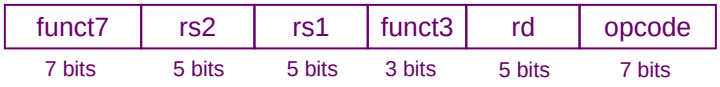
Outline

- **Designing an ISA**
- **Example ISA: RISC-V**
  - **Introduction**
  - **Instruction Formats**
  - **Writing Assembly**
- **RISC-V Simulator (used in Project 1)**

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

1) RISC-V R-format Instructions



- **Instruction fields**
  - **opcode: operation code**
  - **rd: destination register number**
  - **funct3: 3-bit function code (additional opcode)**
  - **rs1: the first source register number**
  - **rs2: the second source register number**
  - **funct7: 7-bit function code (additional opcode)**

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

`add x9,x20,x21`

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

0000 0001 0101 1010 0000 0100 1011 0011_{two} = $015A04B3_{16}$

Logical Operations

- **Instructions for bitwise manipulation**

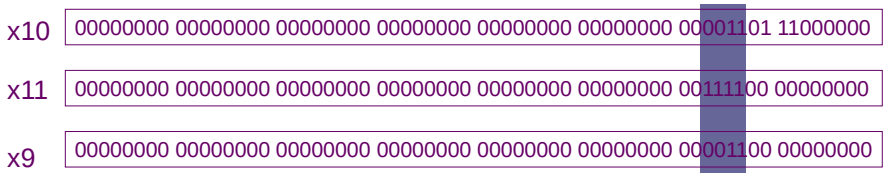
Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

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

AND Operations

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

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



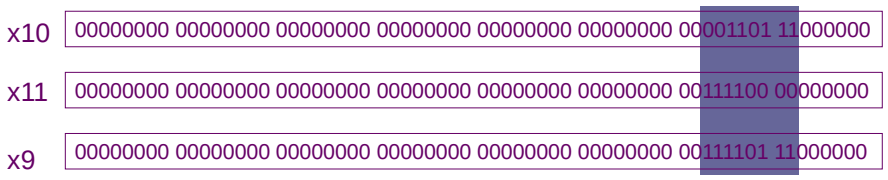
29

29

OR Operations

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

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



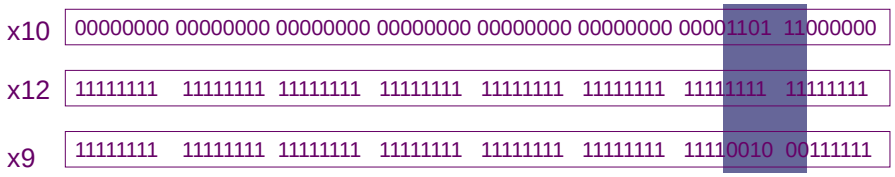
30

30

XOR Operations

- **Differencing operation**
  - **Set some bits to 1, leave others unchanged**

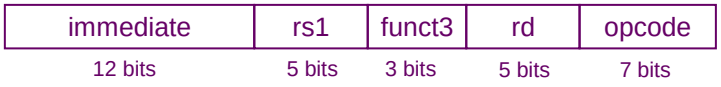
```
xor x9,x10,x12  // NOT operation
```



31

31

2) RISC-V I-format Instructions



- **Immediate arithmetic and Load instructions**
  - **rs1: source or base address register number**
  - **immediate: constant operand, or offset added to base address**
    - **2s-complement, sign extended**
- ***Design Principle 3:* Good design demands good compromises**
  - **Different formats complicate decoding, but allow 32-bit instructions uniformly**
  - **Keep formats as similar as possible**

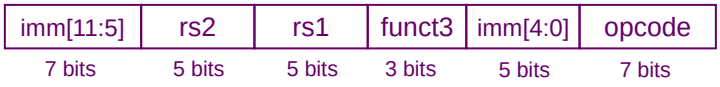
32

32

Shift Operations

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

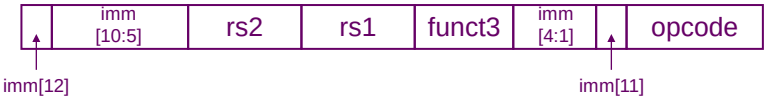
3) RISC-V S-format Instructions



- Different immediate format for Store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

4) RISC-V SB-format - 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

35

35

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit      // branch if b >= a
    addi x22, x22, 1
Exit:
```

36

36

5) RISC-V U-format

- Load upper immediate (`lui`) loads a 20-bit constant into bits 12-31 of a register
- U format:

- For creating 32-bit constants

37

37

32-bit Constants

- 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
```

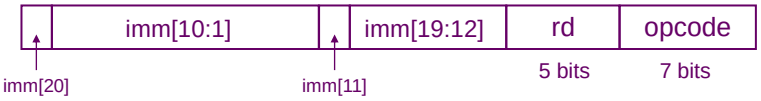

```
addi x19,x19,128  // 0x500
```

38

38

6) RISC-V UB-format - Jump Addressing

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

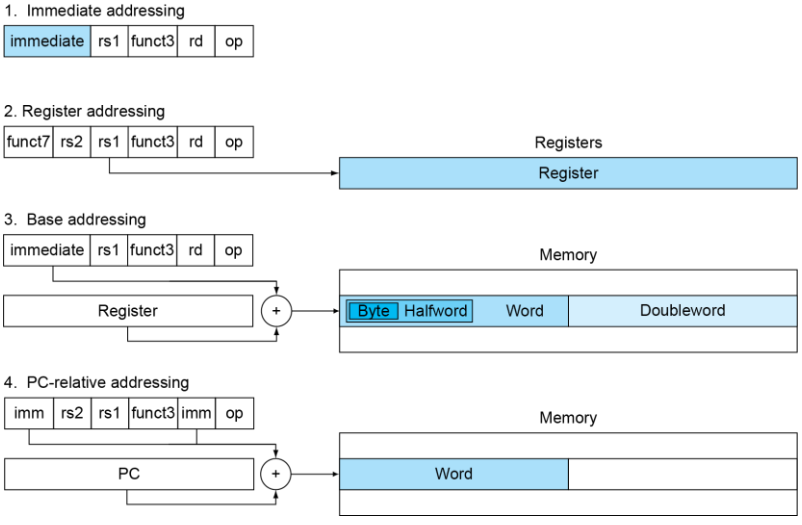


- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

39

39

RISC-V Addressing Summary



40

40

Outline

- Designing an ISA
- Example ISA: RISC-V
  - Introduction
  - Instruction Formats
  - Writing Assembly
- RISC-V Simulator (used in Project 1)

41

41

How to write Assembly

1. Write pseudo or C/Java code
2. Assign variables to registers
3. Write bookkeeping code
   - Beginning of code
     - Change top of stack
     - Push $ra if necessary onto stack
     - Push $sx registers if necessary
   - End of code
     - Pop $sx registers if necessary
     - Pop $ra if necessary
     - Change top of stack
4. Translate body of code

42

42

Register conventions

- Caller saved or *temporary* registers
  - Responsibility of the calling routine to ensure that the register's value is not destroyed inside a call
- Callee saved or *saved* registers
  - Responsibility of the called routine to ensure that the register's value is not destroyed during a call
- Stack pointer
  - $x2
- Return address
  - $x1

43

43

How to Use the Convention

- If subroutine doesn't call any other subroutines
  - Use temporary registers ($tx)
- If subroutine calls other subroutines
  - When register must be maintained across a subroutine call
    - Use saved registers ($sx)
  - When register doesn't need to be maintained across a subroutine call
    - Use temporary registers ($tx)

44

44

Bookkeeping

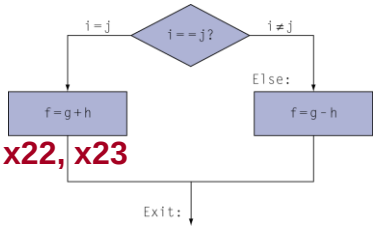
- At the beginning of each subroutine
  - Make room on the stack (increase stack pointer)
  - Push onto stack:
    - $ra register (*if* subroutine will be making any other calls)
    - All saved $sx registers that the subroutine will be using
- At the end of each subroutine
  - Pop from the stack (in reverse order from above):
    - All saved $sx registers that were put on stack
    - $ra register (*if* $ra was put on stack)
  - Delete stack space (decrease stack pointer)

If Statements

- C code:

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

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



- Compiled RISC-V code:

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

Assembler calculates addresses

Loop Statements

- C code:

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

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

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

47

47

Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

48

48

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

49

49

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, x11, x12, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack
    - Actually x5, x6 do NOT really need to be saved (i.e., backed-up); it is done here only to illustrate the Stack concept better

50

50

Leaf Procedure Example

- **RISC-V code:**

```
leaf_example:
  addi sp,sp,-24
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11
  add  x6,x12,x13
  sub  x20,x5,x6
  addi x10,x20,0
  ld   x20,0(sp)
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)
```

// Save x5, x6, x20 on stack

// x5 = g + h
// x6 = i + j
// f = x5 – x6
// Copy f to return register
// Restore x5, x6, x20 from stack

// Return to caller

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

53

53

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

54

54

Non-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
```


C Sort Example

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

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

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

The Procedure Swap

```
swap:
  slli x6,x11,3    // reg x6 = k * 8
  add  x6,x10,x6   // reg x6 = v + (k * 8)
                   // (address of v[k])
  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
```

57

57

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

58

58

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:
```

59

59

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:
```

60

60

The Procedure Body

```
        li   x19,0          // i = 0
for1tst: bge  x19,x11,exit1 // go to exit1 if x19 ≥ x11 (i≥n)
        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:  addi x19,x19,1      // i += 1
        j    for1tst        // branch to test of outer loop
```

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

61

61

The Full Procedure

```
sort:
  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
  ...             // The Procedure Body from previous Slide here
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)   // return to calling routine
```

62

62

Outline

- Designing an ISA
- Example ISA: RISC-V
  - Introduction
  - Instruction Formats
  - Writing Assembly
- RISC-V Simulator (used in Project 1)

63

63

RARS Simulator

- Download simulator:
  - https://github.com/TheThirdOne/rars/releases (download just rars1_4.jar)
- Learn through examples
  - http://dejazzer.com/coen4710/projects.html
- More examples:
  - https://github.com/TheThirdOne/rars/tree/master/examples

64

64

Concluding Remarks

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

65