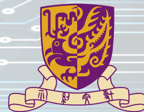
CENG 3420
Computer Organization & Design



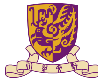
Lecture 03: Instruction Set Architecture

Bei Yu

(Latest update: January 28, 2021)

Spring 2021

Overview



Introduction

Arithmetic & Logical Instructions

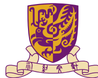
Data Transfer Instructions

Control Instructions

Others

Summary

Overview



Introduction

Arithmetic & Logical Instructions

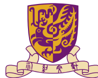
Data Transfer Instructions

Control Instructions

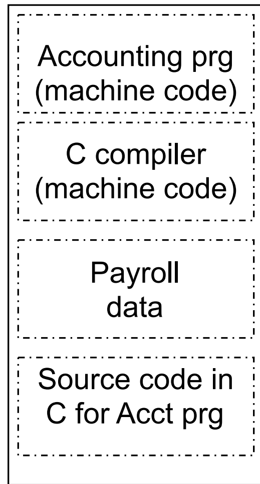
Others

Summary

Two Key Principles of Machine Design



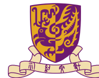
1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data



Stored-Program Concept

- Programs can be shipped as files of binary numbers – binary compatibility
- Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs

Assembly Language Instructions



The language of the machine

- Want an ISA that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost

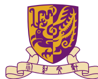
Our target: the RISC-V ISA

- similar to other ISAs developed since the 1980's
- RISC-V is originated from MIPS, the latter of which is used by Broadcom, Cisco, NEC, Nintendo, Sony, ...

Design Goals

Maximize performance, minimize cost, reduce design time (time-to-market), minimize memory space (embedded systems), minimize power consumption (mobile systems)

CISC vs. RISC



Complex Instruction Set Computer (CISC)

Lots of instructions of variable size, very memory optimal, typically less registers.

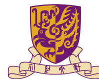
- Intel x86

Reduced Instruction Set Computer (RISC)

Instructions, all of a fixed size, more registers, optimized for speed. Usually called a "Load/Store" architecture.

- RISC-V, LC-3b MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC ...

RISC – Reduced Instruction Set Computer

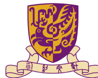


RISC Philosophy

- fixed instruction lengths
- load-store instruction sets
- limited number of addressing modes
- limited number of operations

- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

RISC-V (RISC) Design Principles



Simplicity favors regularity

- fixed size instructions
- small number of instruction formats
- opcode always the first 6 bits

Smaller is faster

- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

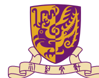
Make the common case fast

- arithmetic operands from the register file (load-store machine)
- allow instructions to contain immediate operands

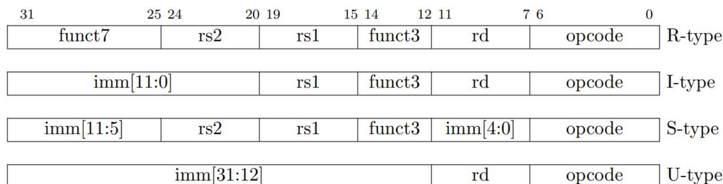
Good design demands good compromises

- For RV32I, 4 base instruction formats (R/I/S/U) and 2 extended instruction formats (B/J)

RISC-V Instruction Fields

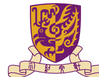


RISC-V fields are given names to make them easier to refer to



opcode	6-bits, opcode that specifies the operation
rs1	5-bits, register file address of the first source operand
rs2	5-bits, register file address of the second source operand
rd	5-bits, register file address of the result's destination
imm	12-bits / 20-bits, immediate number field
funct	3-bits / 10-bits, function code augmenting the opcode

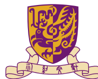
The RISC-V ISA

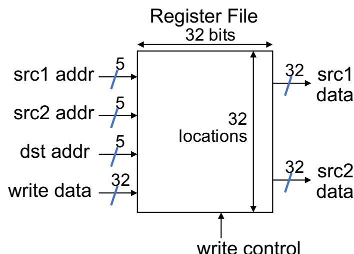


Instruction Categories

- Load and Store instructions
- Bitwise instructions
- Arithmetic instructions
- Control transfer instructions
- Pseudo instructions

RISC-V Register File





- Holds thirty-two 32-bit general purpose registers
- Two read ports
- One write port

Registers are

- Faster than main memory
  - But register files with more locations are slower
  - E.g., a 64 word file may be 50% slower than a 32 word file
  - Read/write port increase impacts speed quadratically
- Easier for a compiler to use
  - `(A*B)-(C*D)-(E*F)` can do multiplies in any order vs. stack
- Can hold variables so that code density improves (since register are named with fewer bits than a memory location)

Aside: RISC-V Register Convention

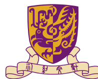
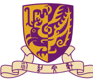


Table: Register names and descriptions

Register Names	ABI Names	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / Alternate link register
x6-7	t1 - t2	Temporary register
x8	s0 / fp	Saved register / Frame pointer
x9	s1	Saved register
x10-11	a0-a1	Function argument / Return value registers
x12-17	a2-a7	Function argument registers
x18-27	s2-s11	Saved registers
x28-31	t3-t6	Temporary registers

History of RISC-V



RISC-V Foundation Incorporated

Privileged Architecture V1.10

Privileged Architecture V1.7

RV32E, RVC V1.9

User ISA V2.0 IMAFD

1st Commercial SoC

1st Rocket tapeout, EOS14, 45nm

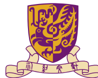
User ISA V1.0
Raven-1 tapeout (28 nm)
RVC

RISC-V ISA project begins

First Linux

2010 2011 2012 2013 2014 2015 2016 2017 2018

Overview



Introduction

Arithmetic & Logical Instructions

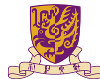
Data Transfer Instructions

Control Instructions

Others

Summary

RISC-V Arithmetic Instructions



- RISC-V assembly language arithmetic statement

```
add     t0, a1, a2
sub     t0, a1, a2
```

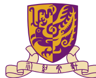
- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`t0, s1, s2`)

```
destination = source1 op source2
```

- Instruction Format (R format)

0x0 / 0x40	0xc	0xb	0	0x5	0x33

RISCV Immediate Instructions



- Small constants are used often in typical code

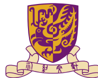
Possible approaches?

- put “typical constants” in memory and load them
- create hard-wired registers (like `zero`) for constants like 1
- have special instructions that contain constants

```
addi sp, sp, 4      # sp = sp + 4
slti t0, s2, 15     # t0 = 1 if s2 < 15
```

- Machine format (I format)
- The constant is kept inside the instruction itself!
- Immediate format limits values to the range -2^{11} to $+2^{11} - 1$

Aside: How About Larger Constants?



- We'd also like to be able to load a 32 bit constant into a register
- For this we must use two instructions

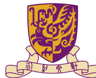
1. A new "load upper immediate" instruction (U-type format, load top 20bits)

```
lui t0, 1010101010101010
```

2. Then must get the lower order bits right, use

```
ori t0, t0, 1010101010101010
```

Aside: How About Larger Constants?



- We’d also like to be able to load a 32 bit constant into a register
- For this we must use two instructions

1. A new “load upper immediate” instruction (U-type format, load top 20bits)

```
lui t0, 1010101010101010
```

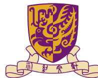
2. Then must get the lower order bits right, use

```
ori t0, t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

RISC-V Shift Operations

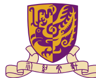


- Need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
slli t2, s0, 8     # t2 = s0 << 8 bits
srli t2, s0, 8     # t2 = s0 >> 8 bits
```

- Instruction Format (I format)
- Such shifts are called logical because they fill with zeros
- Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

RISC-V Logical Operations



There are a number of bit-wise logical operations in the RISC-V ISA

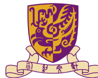
R Format

```
and t0, t1, t2     # t0 = t1 & t2
or  t0, t1, t2     # t0 = t1 | t2
nor t0, t1, t2     # t0 = not(t1 | t2)
```

I Format

```
andi t0, t1, 0xFF00     # t0 = t1 & 0xff00
ori  t0, t1, 0xFF00     # t0 = t1 | 0xff00
```

Overview



Introduction

Arithmetic & Logical Instructions

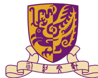
Data Transfer Instructions

Control Instructions

Others

Summary

RISC-V Memory Access Instructions

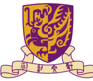


- Two basic data transfer instructions for accessing memory

```
lw  t0, 4(s3)   # load word from memory
sw  t0, 8(s3)   # store word to memory
```

- The data is loaded into (`lw`) or stored from (`sw`) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
- A 12-bit field in RV32I meaning access is limited to memory locations within a region from -4 KB to 4 KB of the address in the base register

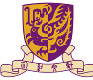
Machine Language – Load Instruction



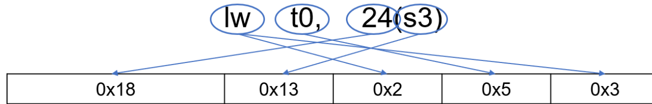
Load/Store Instruction Format (I format):

lw t0, 24(s3)

Machine Language – Load Instruction

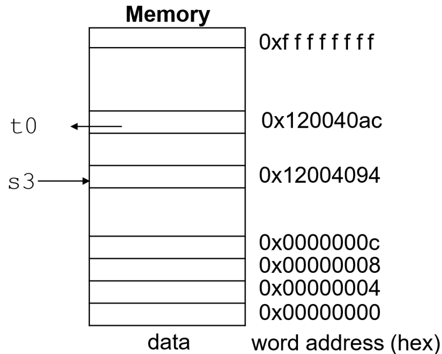


Load/Store Instruction Format (I format):

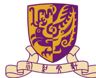


24_{10} + s3 =

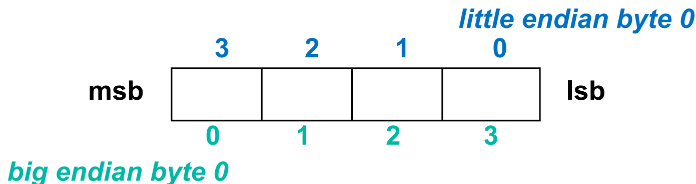
. . . 0001 1000
\+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



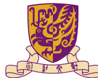
Byte Addresses



- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
- Alignment restriction – the memory address of a word must be on natural word boundaries (a multiple of 4 in RV32I)
- Big Endian: leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  - RISC-V, Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



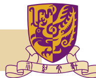
Aside: Loading and Storing Bytes



RISC-V provides special instructions to move bytes

```
lb    t0, 1(s3)   # load byte from memory
sb    t0, 6(s3)   # store byte to memory
```

- What 8 bits get loaded and stored?
- Load byte places the byte from memory in the rightmost 8 bits to the destination register
- Store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory



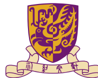
Given following code sequence and memory state:

```
add   s3, zero, zero
lb    t0, 1(s3)
sb    t0, 6(s3)
```

Memory (Data)	Word Address (Decimal)
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0

1. What value is left in `t0`?
2. What word is changed in Memory and to what?
3. What if the machine was little Endian?

Overview



Introduction

Arithmetic & Logical Instructions

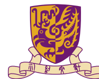
Data Transfer Instructions

Control Instructions

Others

Summary

RISC-V Control Flow Instructions



RISC-V conditional branch instructions:

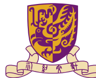
```
bne s0, s1, Lbl     # go to Lbl if s0 != s1
beq s0, s1, Lbl     # go to Lbl if s0 = s1
```

Example

```
        if (i==j) h = i + j;

        bne s0, s1, Lbl1
        add s3, s0, s1
Lbl1:   ...
```

- Instruction Format (B format)
- How is the branch destination address specified ?

In Support of Branch Instructions



- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)?
- For this, we need yet another instruction, `slt`

Set on less than instruction:

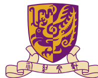
```
slt t0, s0, s1       # if s0 < s1   then
                     # t0 = 1       else
                     # t0 = 0
```

- Instruction format (R format or I format)

Alternate versions of `slt`

```
slti  t0, s0, 25     # if s0 < 25 then t0 = 1 ...
sltu  t0, s0, s1     # if s0 < s1 then t0 = 1 ...
sltiu t0, s0, 25     # if s0 < 25 then t0 = 1 ...
```

Aside: More Branch Instructions



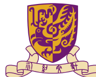
Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `zero` to create other conditions

- less than: `blt s1, s2, Label`

```
slt  t0, s1, s2          # t0 set to 1 if
bne  t0, zero, Label     # s1 < $s2
```

- less than or equal to: `ble s1, s2, Label`
- greater than: `bgt s1, s2, Label`
- great than or equal to: `bge s1, s2, Label`
- Such branches are included in the instruction set as pseudo instructions – recognized (and expanded) by the assembler

Bounds Check Shortcut

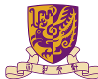


- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu t0, s1, t2         # t0 = 0 if
                        # s1 > t2 (max)
                        # or s1 < 0 (min)
beq  t0, zero, IOOB     # go to IOOB if
                        # t0 = 0
```

- The key is that negative integers in two's complement look like large numbers in unsigned notation.
- Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

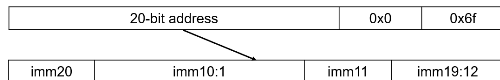
Other Control Flow Instructions

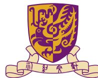


- RISC-V also has an unconditional branch instruction or jump instruction:

```
jal zero, label         # go to label, label can be an
    immediate value
```

- Instruction Format (J Format)
- `J` is a pseudo instruction of unconditional `jal` and it will discard the return address (e.g., `j label`)



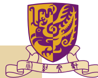


EX-2: Branching Far Away

What if the branch destination is further away than can be captured in 12 bits? Re-write the following codes.

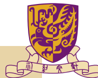
```
beq     s0, s1, L1
```

EX: Compiling a while Loop in C



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

Assume that `i` and `k` correspond to registers `s3` and `s5` and the base of the array save is in `s6`.



EX: Compiling a while Loop in C

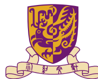
```
while (save[i] == k) i += 1;
```

Assume that i and k correspond to registers s3 and s5 and the base of the array save is in s6.

```
Loop: sll  t1, s3, 2      # Temp reg t1 = i * 4
      add  t1, t1, s6     # t1 = address of save[i]
      lw   t0, 0(t1)      # Temp reg t0 = save[i]
      bne  t0, s5, Exit   # go to Exit if save[i] != k
      addi s3, s3,1       # i = i + 1
      j    Loop           # j is a pseudo instruction for jal
                          # go to Loop
Exit:
```

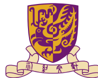
Note: left shift s3 to align word address, and later address is increased by 1

Six Steps in Execution of a Procedure



1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `a0` – `a7`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `s0`–`s11`: 12 value registers for result values
6. Callee returns control to the caller
   - `ra`: one return address register to return to the point of origin

Instructions for Accessing Procedures



We have learnt `jal`, now let's continue

- RISC-V procedure call instruction:

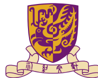
```
jal  ra, label # jump and link,
               # label can be an immediate value
```

- Saves PC + 4 in register `ra` to have a link to the next instruction for the procedure return
- Machine format (J format):
- Then can do procedure return with a

```
jalr x0, 0(ra) # return
```

- Instruction format (I format)

Example of Accessing Procedures

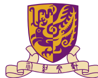


- For a procedure that computes the GCD of two values i (in `t0`) and j (in `t1`):
  `gcd(i,j);`
- The caller puts the i and j (the parameters values) in `a0` and `a1` and issues a

```
    jal ra, gcd        # jump to routine gcd
```

- The callee computes the GCD, puts the result in `s0`, and returns control to the caller using

```
gcd: . . .          # code to compute gcd
     jalr x0, 0(ra)        # return
```



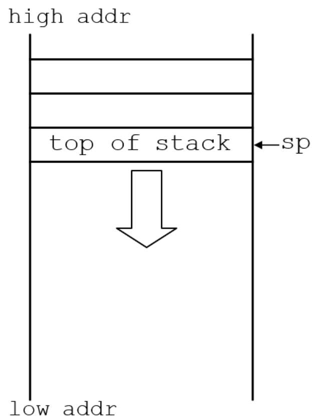
What if the callee needs to use more registers than allocated to argument and return values?

- Use a stack: a last-in-first-out queue
- One of the general registers, `sp`, is used to address the stack
- “grows” from high address to low address
- push: add data onto the stack, data on stack at new `sp`

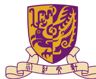
```
sp = sp - 4
```

- pop: remove data from the stack, data from stack at `sp`

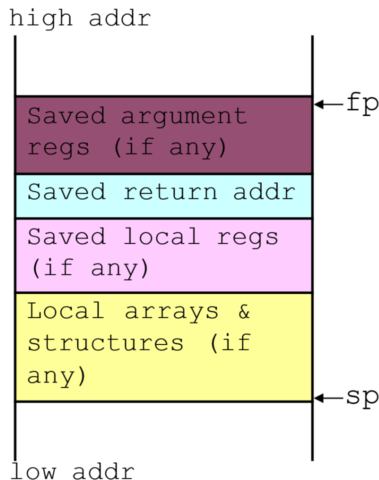
```
sp = sp + 4
```



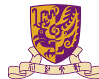
Allocating Space on the Stack



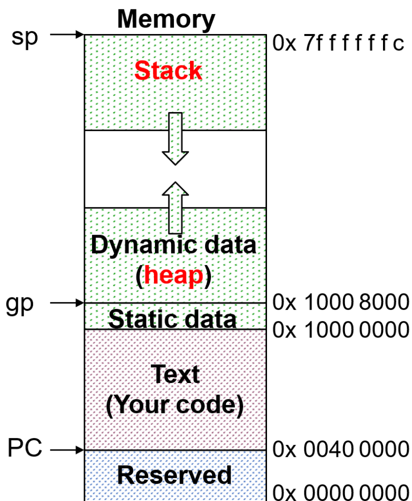
- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
- The frame pointer (`fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
- `fp` is initialized using `sp` on a call and `sp` is restored using `fp` on a return

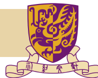


Allocating Space on the Heap



- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
- Allocate space on the heap with `malloc()` and free it with `free()` in C



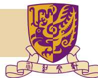


Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

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

Solution:

EX-3: Compiling a C Leaf Procedure



Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

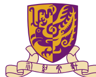
```
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Solution:

Suppose g, h, i, and j are in `a0, a1, a2, a3`

```
leaf_ex:  addi   sp, sp, -8  # make stack room
          sw     t1, 4(sp)   # save t1 on stack
          sw     t0, 0(sp)   # save t0 on stack
          add    t0, a0, a1
          add    t1, a2, a3
          sub    s0, t0, t1
          lw     t0, 0(sp)   # restore t0
          lw     t1, 4(sp)   # restore t1
          addi   sp, sp, 8   # adjust stack ptr
          jalr   zero, 0(ra)
```

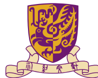
Nested Procedures



- Nested Procedure: call other procedures
- What happens to return addresses with nested procedures?

```
int rt_1 (int i)
{
    if (i == 0) return 0;
    else return rt_2(i-1);
}
```

Nested procedures (cont.)



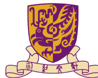
```
caller: jal  rt_1
next:   . . .

rt_1:   bne  a0, zero, to_2
        add  s0, zero, zero
        jalr zero, 0(ra)
to_2:   addi a0, a0, -1
        jal  ra, rt_2
        jalr zero, 0(ra)

rt_2:   . . .
```

- On the call to `rt_1`, the return address (next in the caller routine) gets stored in `ra`.

Question:

What happens to the value in `ra` (when `a0 != 0`) when `to_2` makes a call to `rt_2`?

Compiling a Recursive Procedure



A procedure for calculating factorial

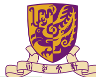
```
int fact (int n)
{
    if (n < 1) return 1;
    else return (n * fact (n-1));
}
```

- A recursive procedure (one that calls itself!)

```
fact (0) = 1
fact (1) = 1 * 1 = 1
fact (2) = 2 * 1 * 1 = 2
fact (3) = 3 * 2 * 1 * 1 = 6
fact (4) = 4 * 3 * 2 * 1 * 1 = 24
. . .
```

- Assume n is passed in `a0`; result returned in `s0`

Compiling a Recursive Procedure (cont.)



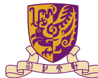
```
fact: addi  sp, sp, -8       # adjust stack pointer
      sw    ra, 4(sp)        # save return address
      sw    a0, 0(sp)        # save argument n
      slti  t0, a0, 1        # test for n < 1
      beq   t0, zero, L1     # if n >= 1, go to L1
      addi  s0, zero, 1      # else return 1 in s0
      addi  sp, sp, 8        # adjust stack pointer
      jalr  zero, 0(ra)      # return to caller
L1:   addi  a0, a0, -1       # n >= 1, so decrement n
      jal   ra, fact         # call fact with (n-1)
                             # this is where fact returns
bk_f: lw    a0, 0(sp)        # restore argument n
      lw    ra, 4(sp)        # restore return address
      addi  sp, sp, 8        # adjust stack pointer
      mul   s0, a0, s0       # s0 = n * fact(n-1)
      jalr  zero, 0(ra)      # return to caller
```

Note: `bk_f` is carried out when `fact` is returned.

Question:

Why we don't load `ra, a0` back to registers?

Overview



Introduction

Arithmetic & Logical Instructions

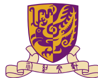
Data Transfer Instructions

Control Instructions

Others

Summary

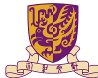
Atomic Exchange Support



- Need hardware support for synchronization mechanisms to avoid data races where the results of the program can change depending on how events happen to occur
- Two memory accesses from different threads to the same location, and at least one is a write
- Atomic exchange (atomic swap): interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
- Implementing an atomic exchange would require both a memory read and a memory write in a single, uninterruptable instruction.
- An alternative is to have a pair of specially configured instructions

```
lr.w  t1, 0(s1)     # Load-Reserved
sc.w  t0, 0(s1)     # Store-Conditional
```

Automic Exchange with `lr` and `sc`

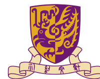


- `lr` and `sc` can construct a lock-free program
- `lr.w` loads a word from the memory, and registers a reservation set - a set of bytes that subsumes the bytes in the addressed word
- `sc.w` conditionally writes a word. The `sc.w` succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the `sc.w` succeeds, the instruction writes the word to the memory, and it writes zero to the `rd`. If the `sc.w` fails, the instruction does not write to the memory, and it writes a nonzero value to `rd`. bytes being written.

Example:

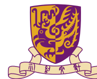
```
    # At the beginning, a0 saves the memory base address
    # a1 saves the expected value
    # a2 saves another expected value
cas:
    lr.w t0, 0(a0)          # read the original value
    bne t0, a1, fail        # if a mismatch occurs, go to fail
    sc.w a0, a2, 0(a0)      # try to update
    jalr zero, 0(ra)        # return
fail:
    li a0, 1                # set the fail flag
    jalr zero, 0(ra)        # return
```

The C Code Translation Hierarchy



C program → compiler → assembly code → assembler → object code → linker (with library routines) → executable → loader → memory

Compiler Benefits



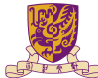
- Comparing performance for bubble (exchange) sort
- To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

The un-optimized code has the best CPI∗, the O1 version has the lowest instruction count, but the O3 version is the fastest.

gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

∗CPI: clock cycles per instruction

Overview



Introduction

Arithmetic & Logical Instructions

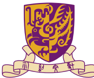
Data Transfer Instructions

Control Instructions

Others

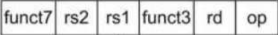
Summary

Addressing Modes Illustrated

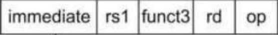


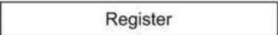
1. Immediate addressing

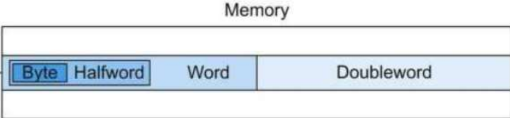
2. Register addressing



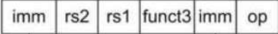
3. Base addressing

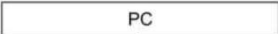


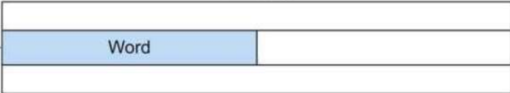




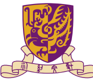
4. PC-relative addressing







RISC-V Organization So Far



Processor

Register File

src1 addr 5
src2 addr 5
dst addr 5
write data 32

32registers
(x0 –x31)

32 bits

src1 data 32
src2 data 32

branch offset 32
PC 32
32 Add
4 32
Add 32
32

Fetch
PC = PC+4
Exec
Decode

32 ALU 32
32

read/write addr 32
read data 32
write data 32

Memory

4	5	6	7
0	1	2	3

32 bits

byte address
(big Endian)

1…1100
0…1100
0…1000
0…0100
0…0000

2^{30} words

word address
(binary)