CENG 3420
Computer Organization & Design

Lecture 04: Control Instruction

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Textbook: Chapters 2.8 – 2.11)

Spring 2023

Overview

Introduction

RISC-V Instruction Fields

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

<table>
<tr><th>31</th><th>30   25</th><th>24   21</th><th>20</th><th>19   15</th><th>14   12</th><th>11   8</th><th>7</th><th>6   0</th><th></th></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-type</td></tr>
</table>

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

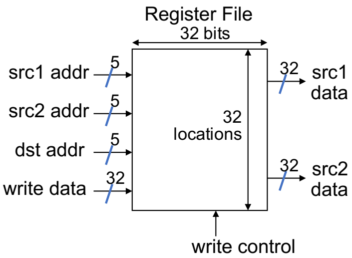
imm 12-bits / 20-bits, immediate number field

funct 3-bits / 10-bits, function code augmenting the opcode

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

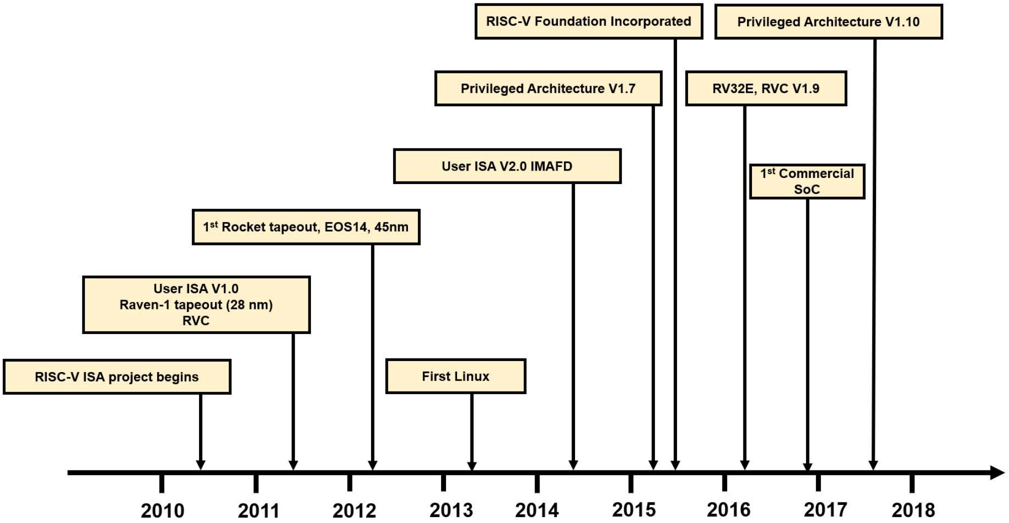
- Faster than main memory
  - But register files with more locations are slower
  - E.g., a 64 word file may be 50% slower than a 32 word file
  - Read/write port increase impacts speed quadratically
- Easier for a compiler to use
  - `(A*B)-(C*D)-(E*F)` can do multiplies in any order vs. stack
- Can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



Control Instructions

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 ?

```
.globl _start

.text
_start:
        li a0, 1
        li a1, 1
        li t0, 20
        li t1, 23
        bne t0, t1, inst1
        addi a0, a0, 1
        beq t0, t1, inst2
inst1:  addi a0, a0, 2
        bne t0, zero, end
inst2:  addi a0, a0, 3
end:    sub a0, a0, a1
```

RARS example: beq

- What is the final value of a0?

RISC-V Control Flow Instructions

```
.globl _start

.text
_start:
        li a0, 1
        li a1, 1
        li t0, 20
        li t1, 23
        bne t0, t1, inst1
        addi a0, a0, 1
        beq t0, t1, inst2
inst1:  addi a0, a0, 2
        bne t0, zero, end
inst2:  addi a0, a0, 3
end:    sub a0, a0, a1
```

RARS example: beq

- What is the final value of a0?
- a0 = 0x2

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 ...
```

In Support of Branch Instructions

```
.globl _start

.text
_start:
            li a0, 1
            li t0, 20
            li t1, 23
            slt a1, t0, t1
            beq a0, a1, inst1
            addi a0, a0, 2
inst1:      addi a0, a0, 3
```

RARS example: slt

- What is the final value of a0?

In Support of Branch Instructions

```
.globl _start

.text
_start:
        li a0, 1
        li t0, 20
        li t1, 23
        slt a1, t0, t1
        beq a0, a1, inst1
        addi a0, a0, 2
inst1:  addi a0, a0, 3
```

RARS example: slt

- What is the final value of a0?
- a0 = 0x4

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
```

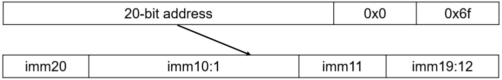
- The key is that negative integers in two's complement look like large numbers in unsigned notation.
- Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

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`)



pc := pc + sign extended(imm20 << 1)

In Support of Branch Instructions

```
.globl _start

.text
_start:
            li a0, 1
            li t0, 20
            jal ra, loop
loop:
            addi a0, a0, 1
            beq a0, t0, end
            j loop # j is a pseudo instruction for jal
end:        addi a0, a0, 1
```

RARS example: jal

- What is the final value of a0?

In Support of Branch Instructions

```
.globl _start

.text
_start:
        li a0, 1
        li t0, 20
        jal ra, loop
loop:
        addi a0, a0, 1
        beq a0, t0, end
        j loop # j is a pseudo instruction for jal
end:    addi a0, a0, 1
```

RARS example: jal

- What is the final value of a0?
- a0 = 0x15

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`: for 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

```
.globl _start

.text
_start:
        li a0, 20
        li a1, 23
        # we call a function: add_two_numbers,
        # and put the result in t1
        jal ra, add_two_numbers
        addi t1, a2, 0 # a2 = add_two_numbers(a0, a1)
        j end

add_two_numbers:
        mv a3, a0 # mv is a pseudo instruction for addi
        mv a4, a1 # equal to "addi a4, a1, 0"
        add a2, a3, a4
        jalr zero, 0(ra)

end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: accessing a procedure with jal & jarl

- What is the final value of t1?

Example of Accessing Procedures

```
.globl _start

.text
_start:
        li a0, 20
        li a1, 23
        # we call a function: add_two_numbers,
        # and put the result in t1
        jal ra, add_two_numbers
        addi t1, a2, 0 # a2 = add_two_numbers(a0, a1)
        j end

add_two_numbers:
        mv a3, a0 # mv is a pseudo instruction for addi
        mv a4, a1 # equal to "addi a4, a1, 0"
        add a2, a3, a4
        jalr zero, 0(ra)

end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: accessing a procedure with jal & jarl

- What is the final value of t1?
- t1 = 0x2c

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
```

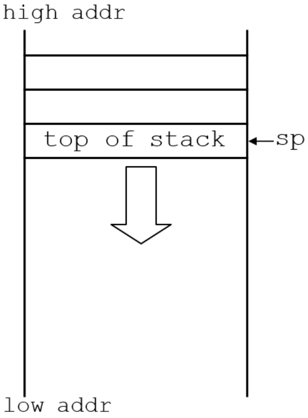
What if the callee needs to use more registers than allocated to argument and return values?

- 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
```

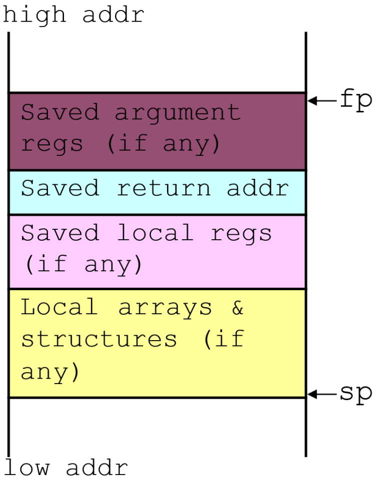
- pop: remove data from the stack, data from stack at `sp`

```
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 Stack

```
.globl _start

.text
_start:
        li a0, 20
        li a1, 23
        # we call a function: add_two_numbers,
        # and put the result in t1
        jal ra, add_two_numbers
        addi t1, a2, 0 # a2 = add_two_numbers(a0, a1)
        j end

add_two_numbers:
        addi sp, sp -8 # we assign 8x4 bytes in the stack
                       # stack: top (high address) -> bottom (low address)
        sw a0, 4(sp)   # we save arguments in the stack
        sw a1, 0(sp)
        add a2, a0, a1 # the a0 and a1 can be used directly since the
                       # original values of a0 and a1 are saved in the stack
        lw a0, 4(sp)   # we restore arguments
        lw a1, 0(sp)
        addi sp, sp, 8 # NOTICE: we need to free the stack we have allocated!
        jalr zero, 0(ra)

end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: allocating space on the stack

- What is the final value of t1?

Allocating Space on the Stack

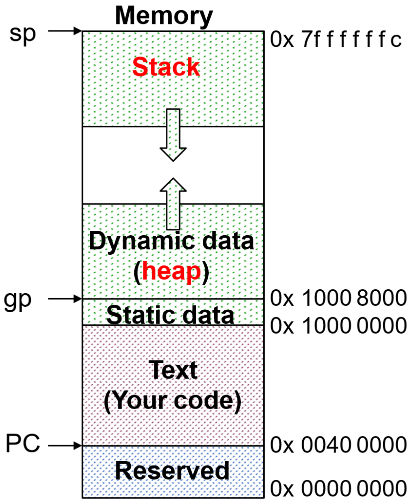
```
.globl _start

.text
_start:
        li a0, 20
        li a1, 23
        # we call a function: add_two_numbers,
        # and put the result in t1
        jal ra, add_two_numbers
        addi t1, a2, 0 # a2 = add_two_numbers(a0, a1)
        j end

add_two_numbers:
        addi sp, sp -8 # we assign 8x4 bytes in the stack
                       # stack: top (high address) -> bottom (low address)
        sw a0, 4(sp)   # we save arguments in the stack
        sw a1, 0(sp)
        add a2, a0, a1 # the a0 and a1 can be used directly since the
                       # original values of a0 and a1 are saved in the stack
        lw a0, 4(sp)   # we restore arguments
        lw a1, 0(sp)
        addi sp, sp, 8 # NOTICE: we need to free the stack we have allocated!
        jalr zero, 0(ra)

end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: allocating space on the stack

- What is the final value of t1?
- t1 = 0x2c

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



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:

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?

Compiling a Recursive Procedure (cont.)

```
.globl _start
.text
_start: li a0, 20
        li a1, 23
        jal ra, func # we call a function: func
                     # func implements (a0 x 2 + a1)
                     # and put the result in t1
        addi t1, a2, 0 # a2 = func(a0, a1)
        j end
func:   addi sp, sp -12
        sw ra, 8(sp)
        sw a0, 4(sp)
        sw a1, 0(sp)
        slli a0, a0, 1
        jal ra, add_two_numbers # add_two_numbers implements (a0 + a1)
        lw ra, 8(sp)
        lw a0, 4(sp)
        lw a1, 0(sp)
        addi sp, sp, 12
        jalr zero, 0(ra)
add_two_numbers: addi sp, sp -8 # we assign 8x4 bytes in the stack
                                # stack: top (high address) -> bottom (low address)
        sw a0, 4(sp)   # we save arguments in the stack
        sw a1, 0(sp)
        add a2, a0, a1 # the a0 and a1 can be used directly since the
                       # original values of a0 and a1 are saved in the stack
        lw a0, 4(sp)   # we restore arguments
        lw a1, 0(sp)
        addi sp, sp, 8 # NOTICE: we need to free the stack we have allocated!
        jalr zero, 0(ra)
end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: compiling a recursive procedure

- What is the final value of t1?

Compiling a Recursive Procedure (cont.)

```
.globl _start
.text
_start: li a0, 20
        li a1, 23
        jal ra, func # we call a function: func
                     # func implements (a0 x 2 + a1)
                     # and put the result in t1
        addi t1, a2, 0 # a2 = func(a0, a1)
        j end
func:   addi sp, sp -12
        sw ra, 8(sp)
        sw a0, 4(sp)
        sw a1, 0(sp)
        slli a0, a0, 1
        jal ra, add_two_numbers # add_two_numbers implements (a0 + a1)
        lw ra, 8(sp)
        lw a0, 4(sp)
        lw a1, 0(sp)
        addi sp, sp, 12
        jalr zero, 0(ra)
add_two_numbers: addi sp, sp -8 # we assign 8x4 bytes in the stack
                                # stack: top (high address) -> bottom (low address)
        sw a0, 4(sp)   # we save arguments in the stack
        sw a1, 0(sp)
        add a2, a0, a1 # the a0 and a1 can be used directly since the
                       # original values of a0 and a1 are saved in the stack
        lw a0, 4(sp)   # we restore arguments
        lw a1, 0(sp)
        addi sp, sp, 8 # NOTICE: we need to free the stack we have allocated!
        jalr zero, 0(ra)
end:
        # we add t1 again
        addi t1, t1, 1
```

RARS example: compiling a recursive procedure

- What is the final value of t1?
- t1 = 0x40

Atomic

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
```

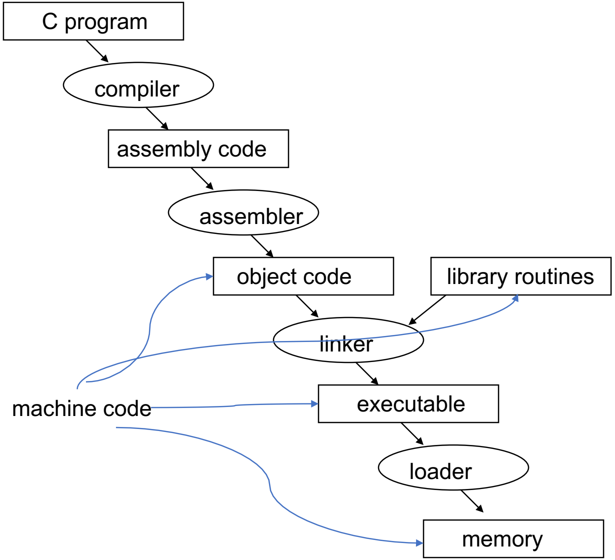
Automic Exchange with `lr` and `sc`

- `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.

Examples:

```
# 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



- 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

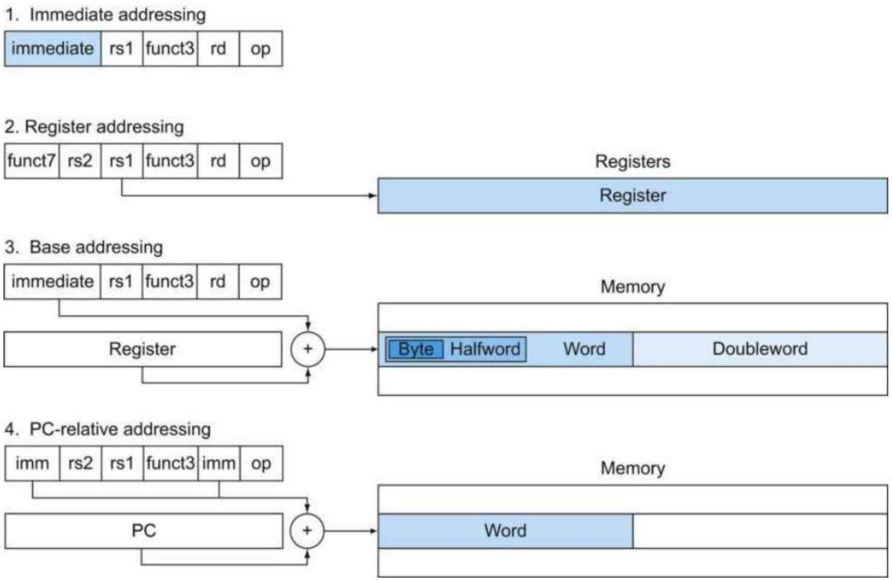
The un-optimized code has the best CPI[1], the O1 version has the lowest instruction count, but the O3 version is the fastest.

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

[1]CPI: clock cycles per instruction

Summary

Addressing Modes Illustrated



RISC-V Organization So Far