CMPE110 Lecture 07
Instruction Set Architecture

Heiner Litz

https://canvas.ucsc.edu/courses/19290

Announcements

- Questions and Responsibilities
  - Lecture content → Instructor (Heiner)
  - Homework questions → TAs
- Piazza Questions
  - Many redundant questions → Use search option
  - We will not debug your personal machine problems
  - Read the manual ("Why does perf APP not work?")
- DRC Students: Please let me know if you do not need **Quiz** accommodation
- Enrollment/Waitlist pretty stable

Review

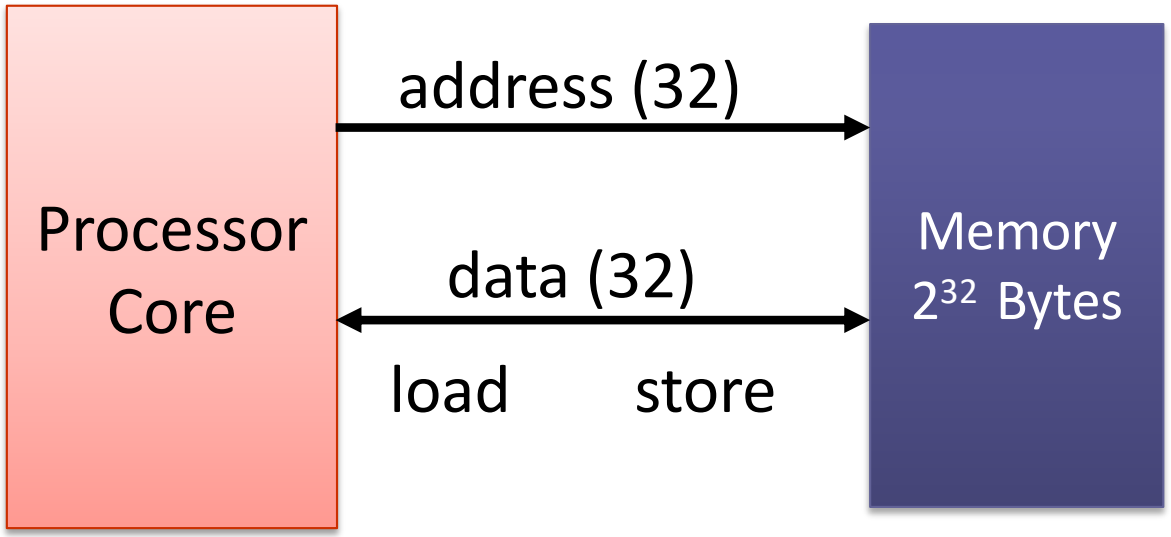
Memory Data Transfer

Data transfer instructions move data to and from memory

- Load moves data from a memory location to a register
- Store moves data from a register to a memory location

All memory access happens through loads and stores

- Floating-point loads and stores for accessing FP registers



Data Transfer Instructions: Loads

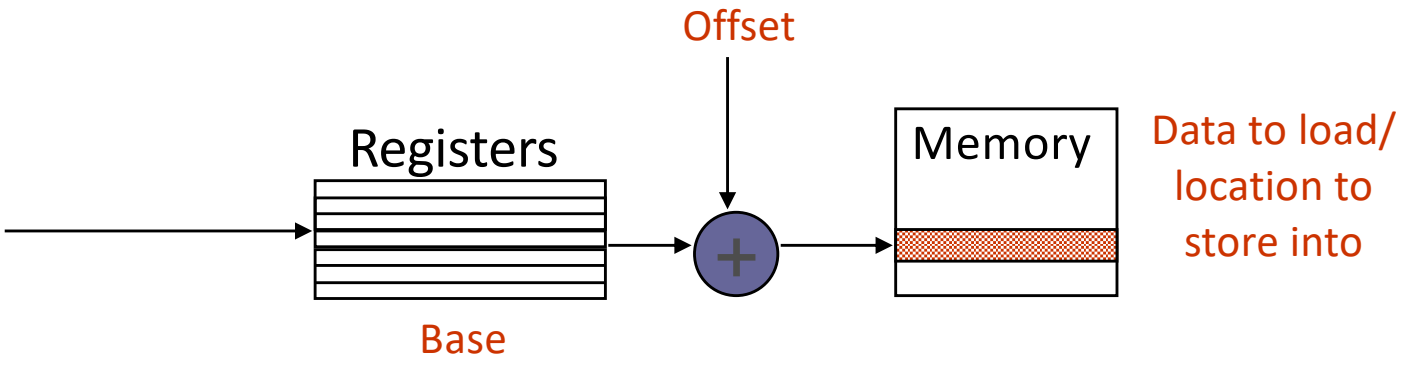
Data transfer instructions have three parts

- Operator name (defines the transfer size as well)
- Destination register
- Base register address and constant offset

```
ld dst, offset(base)
```

- Offset value is a 12-bit signed constant (immediate)

Displacement Addressing Mode



Effective address is a byte addresses

- Must be aligned to words, half-words, & bytes
- More on this later

Loading Data Example

Consider the C example: `a = b + *c;`

Assume a in registers x1, b in x2

Assume c in memory, address stored in x3

`ld` instruction:

```
Ld xt0, 0(x3)          # xt0 = Memory[c]
                       # xt0 is temp reg
Add x1, x2, xt0        # a = b + *c
```

Accessing Arrays

Arrays are really pointers to the base address in memory

Address of element A[0]

Use offset value to indicate which index

Note: addresses are in bytes, so multiply by the size of the element

Unlike C, assembly does not do pointer arithmetic for you!

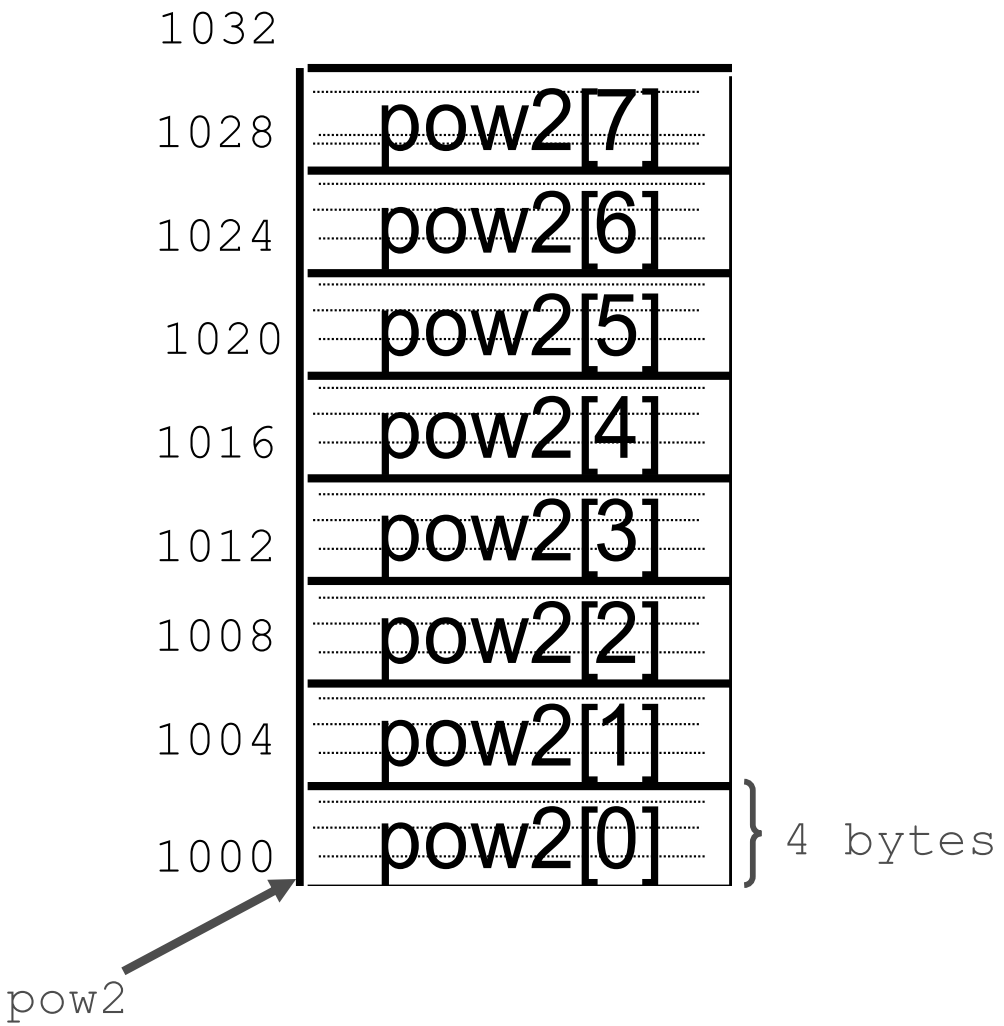
Consider the integer array where `pow2` is the base address

With this compiler on this architecture, each int requires 4 bytes

The data to be accessed is at index 5: `pow2[5]`

Then the address from memory is `pow2 + 5 * 4`

Array Memory Diagram



Array Example

Example: `a = b + pow2[7];`

`x3 = 1000`

`lw` instruction with offset:

```
Ld xt0, 28(x3)    # xt0 = Memory[pow2[7]]
Add x2, x1, xt0   # a = b + pow2[7]
```

Storing Data

Store is the reverse of load

And identical in address generation

Copy data from the source register to an address in memory

```
sd src, offset(base)
```

Offset value is a 12-bit signed constant

Storing Data Example

Example: `*a = b + c;`

Assume a (address) in x3, b in x1, c in x2

`sd` instruction:

```
add xt0, x1, x2          # $t0 = b + c
sd xt0, 0(x3)            # Memory[s0] = b + c
```

Storing to an Array

Example: `a[3] = b + c;`

Assume b in x2, c in x3, a in x4

sd instruction with offset

```
add xt0, x2, x3       # $t0 = b + c
sd xt0, 12(x4)        # Memory[a[3]] = b + c
```

Question: Why 12 and not 3?

Indexed Array Storage

Example: `a[i] = b + c;`

Assume i in x3, b in x1, c in x2, a in x4

Address generation + store:

```
Add xt0, x1, x2      # $t0 = b + c
Sll xt1, x3, 2       # $t1 = 4 * i
Add xt2, x4, xt1     # $t2 = a + 4*i
sd  xt0, 0(xt2)      # Memory[a[i]] = b+c
```

Interface to I/O

Load/stores provide interface to memories

How about interface to I/O devices?

Huge variety in I/O devices

- Printer, USB camera, network interface, hard disk….
- Huge variety in functionality and performance requirements

Special I/O instructions for each type of device

- What would be the problem with this?

Interface to I/O

Break the problem in two

- Communicating bits to the I/O device
- Executing operations based on these bits

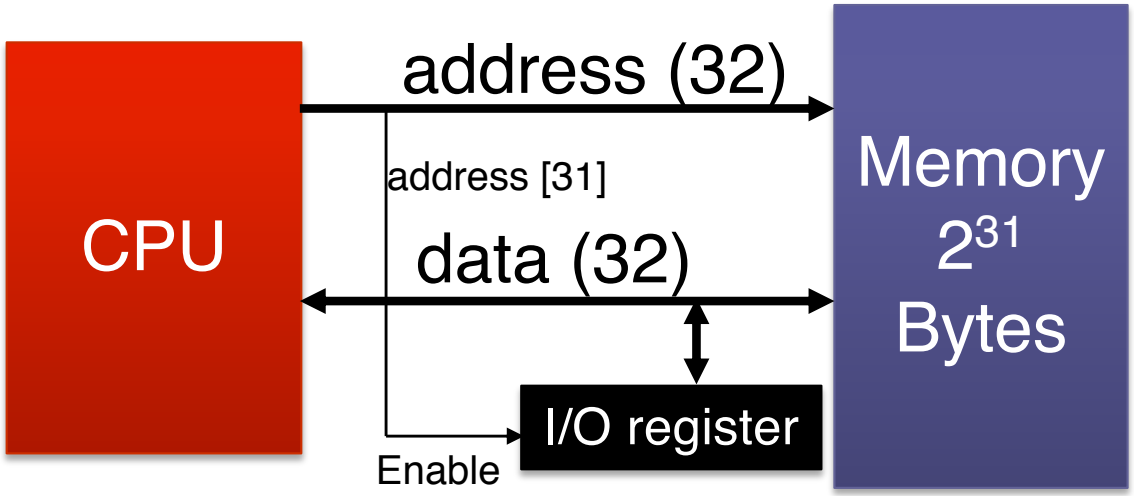
Register/memory based interface to I/O

- Every device includes some registers and memory
- Reading/writing these registers communicates bits
  - Similar across all I/O devices
- Every device has it own protocol of what these bits mean
  - Check status, read command, send command, …
  - Software must know this protocol (device driver)

Memory Mapped I/O: Preview

Loads/stores can be used to move bits to/from I/O devices

- A load moves data from a an I/O device register to a CPU register
- A store moves data from a CPU register to a I/O device register



I/O register at address 0x80000000

Changing Flow Control

Changing Control Flow

Programs have complex flow control

- If-then-else and case statements
- Loops (if, while, …)
- Function/method/procedure calls

Control flow operations: two types

- Conditional branch instructions are known as branches
- Unconditional changes in the control flow are called jumps

The target of the branch/jump is a label

- Label identifies a location in a program

RISC-V Conditional Branch

The simplest conditional test is the `beq` instruction for equality

```
beq reg1, reg2, label
```

Consider the code

```
        if (a == b) goto L1;
        a = a + c;
L1:     // Continue
```

Use the `beq` instruction

```
        beq x1, x2, label   (label is an immediate added to PC)
        add x1, x1, x2
        …
L1:     # Continue
```

RISC-V Unconditional Jump

The `j` instruction jumps to a label

```
j label
```

This is essentially a goto statement

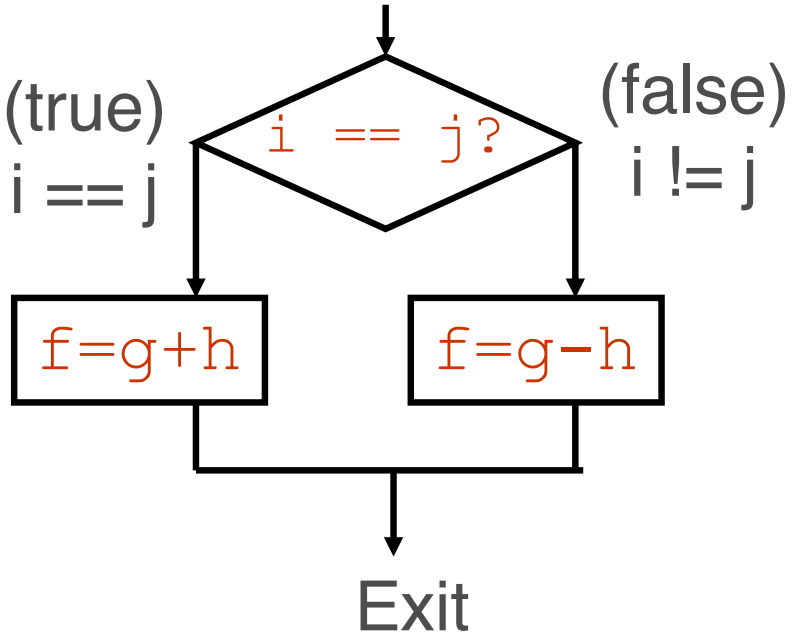
PC relative addressing (what is a label?)

- Labels have to be 0s and 1s as well ☺
- Assembler translates labels into immediate
- Immediate is added to PC
- Enables jump within (+/-) 2^18 of current PC

If-then-else Example

Consider the code

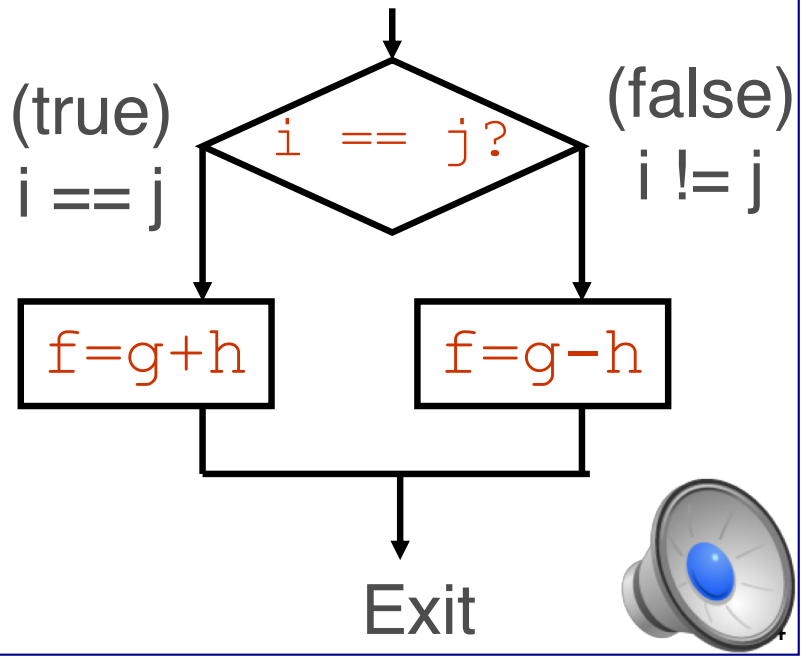
```
if (i == j) f = g + h;
else f = g - h;
```



If-then-else Example

Create labels and use equality instruction

```
        beq x3, x4, True     # Branch if i == j
False:  subu x0, x1, x2      # f = g - h
        j Exit               # Go to Exit
True:   add x0, x1, x2       # f = g + h
Exit:
```



RISC-V Jumps & Branches

Instruction	Example	Meaning
jump	j L	goto L
jump register	jr x1	goto value in x1
jump and link	jal L	goto L and set xra
jump and link register	jalr x1	goto x1 and set xra

xra = return address register

branch equal	beq x1, x2, L	if (x1 == xs2) goto L
branch not eq	bne x1, x2, L	if (x1 != x2) goto L
branch l.t. 0	bltz x1, L	if (x1 < 0) goto L
branch l.t./eq 0	blez x1, L	if (x1 <= 0) goto L
branch g.t. 0	bgtz x1, L	if (x1 > 0) goto L
branch g.t./eq 0	bgez x1, L	if (x1 >= 0) goto L

Branch on Other Comparisons

How do we branch on other comparisons?

<, >, ≤, ≥

Two instruction solution

A comparison instruction with binary outputs (e.g., slt = set less than)

An equality/inequality branch on the comparison output

Consider the following C code

```
if (f < g) goto Less;
```

Solution

```
slt xt0, xs0, xs1          # xt0 = 1 if xs0 < xs1
bne xt0, xzero, Less       # Goto Less if xt0 != 0
```

RISC-V Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt x1, x2, x3`	x1 = (x2 < x3)	comp less than signed
set less than imm	`slti x1, x2, 100`	x1 = (x2 < 100)	comp w/const signed
set less than uns	`sltu x1, x2, x3`	x1 = (x2 < x3)	comp < unsigned
set l.t. imm. uns	`sltiu x1, x2, 100`	x1 = (x2 < 100)	comp < const unsigned

$$A < B = B > A$$
$$!(A < B) = A \geq B$$

C

```
if (a < 8)
```

Assembly

```
slti xv0,xa0,8              # xv0 = a < 8
beq  xv0,xzero, Exceed      # goto Exceed if xv0 == 0
```

While loop in C

Consider a `while` loop

```
while (A[i] == k)
      i = i + j;
```

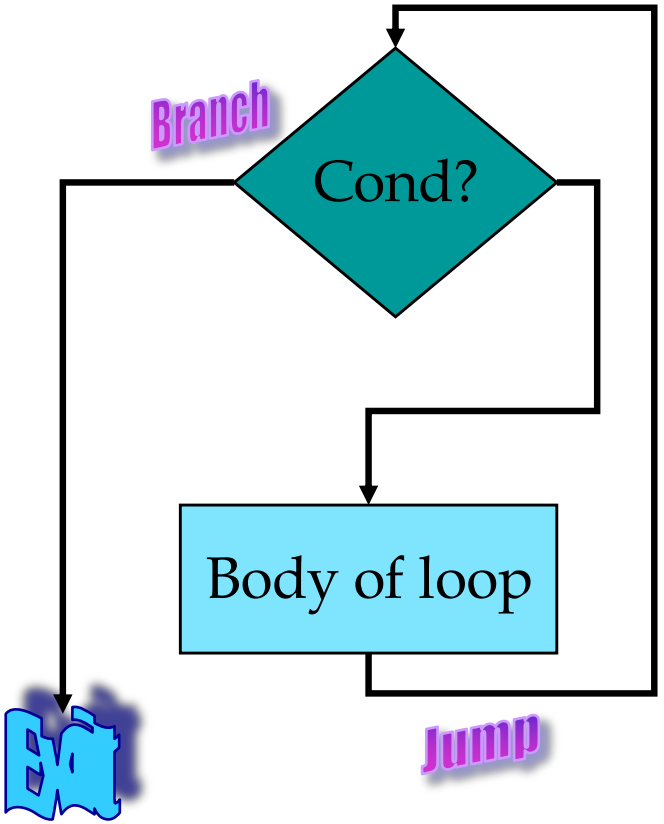
Any idea?

assume `i in x0, j in x1, k in x2, &A[0] in x3`

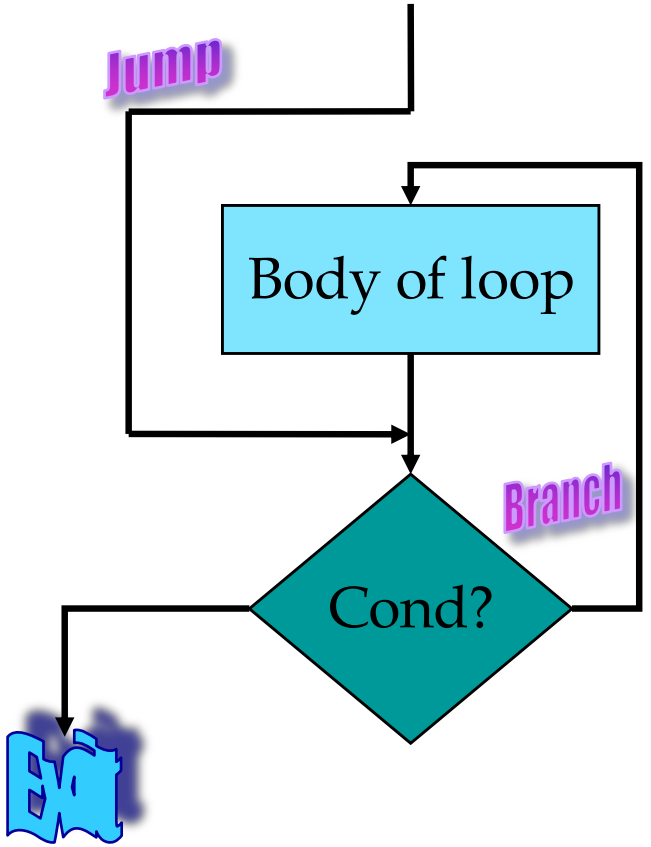
```
Loop:  sll xt0, x0, 2          # xt0 = 4 * i
       addu xt1, xt0, x3       # xt1 = &(A[i])
       lw xt2, 0(xt1)          # xt2 = A[i]
       bne xt2, x2, Exit       # goto Exit if !=
       addu x0, x0, x1         # i = i + j
       j Loop                  # goto Loop
Exit:
```

Improve Loop Efficiency

Two branches/iteration:



Better structure:



Improved Loop Solution

Remove extra jump from loop body

```
        j Cond                  # goto Cond
Loop:   addu xs0, xs0, xs1      # i = i + j
Cond:   sll xt0, xs0, 2         # xt0 = 4 * i
        addu xt1, xt0, xs3      # xt1 = &(A[i])
        lw xt2, 0(xt1)          # xt2 = A[i]
        beq xt2, xs2, Loop      # goto Loop if ==
Exit:
```

Reduced loop from 6 to 5 instructions

Even small improvements important if loop executes many tim