Where we are going (today)

Bits store whatever you want, including integers. How do we manipulate sequences of bits?

10110000

Q: How do we arrange bits in the memory of the computer? (why do we care? we want the computer to store many individual numbers)
A: bytes and words

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110000</td>
<td>00001110</td>
<td>01000010</td>
<td>11110001</td>
</tr>
</tbody>
</table>

partway done

Q: How do we name or refer to all those individual numbers in memory?
A: addresses and pointers

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110000</td>
<td>00001110</td>
<td>01000010</td>
<td>11110001</td>
</tr>
</tbody>
</table>
CS 2630
Computer Organization

Meeting 3: memory organization and addresses
Brandon Myers
University of Iowa
organizes its bits as **bytes and words**
Organize bytes into machine **words**

(note address in hex)

<table>
<thead>
<tr>
<th>Address</th>
<th>32-bit words</th>
<th>64-bit words</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td><strong>1A</strong></td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS has 32-bit addresses

What is the largest number of bytes the memory of a MIPS computer can have?
Where we are going (today)

Bits store whatever you want, including integers. How do we manipulate sequences of bits?

10110000

Q: How do we arrange bits in the memory of the computer? (why do we care? we want the computer to store many individual numbers)
A: bytes and words

Q: How do we name or refer to all those individual numbers in memory?
A: addresses and pointers

(done for now)

next
Looking at memory as words

<table>
<thead>
<tr>
<th>0000</th>
<th>0004</th>
<th>0008</th>
<th>000C</th>
<th>0010</th>
<th>0014</th>
<th>0018</th>
<th>001C</th>
<th>0020</th>
<th>0024</th>
<th>0028</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Storing values

- store the value integer $15_{10}$ at address 0x0008
Storing values

- store the **value** integer $15_{10}$ at **address** 0x0014
- store the **value** integer $20_{10}$ at **address** 0x0000
Addresses can be values, too!

- store the **value** integer $15_{10}$ at **address** 0x0014
- store the **value** integer $20_{10}$ at **address** 0x0000
- store the **value** **address** 0x0014 at **address** 0x001C
Addresses can be values, too!

- store the **value** integer $15_{10}$ at **address** 0x0014
- store the **value** integer $20_{10}$ at **address** 0x0000
- store the **value** **address** 0x0014 at **address** 0x001C
  - we say the word at **address** 0x001C is a **pointer** to the integer at **address** 0x0014
Addresses can be values, too!

- store the value integer \(15_{10}\) at address \(0x0014\)
- store the value integer \(20_{10}\) at address \(0x0000\)
- store the value address \(0x0014\) at address \(0x001C\)
  - we say the word at address \(0x001C\) is a pointer to the integer at address \(0x0014\)
- store the value address \(0x001C\) at address \(0x0028\)
  - we say the word at address \(0x0028\) is a pointer to a pointer to an integer
int[] arr = new int[3];
int[] arr = new int[3];
arr[0] = 13;

How arrays look in memory
int[] arr = new int[3];
arr[0] = 13;
arr[1] = 10;
How arrays look in memory

```java
int[] arr = new int[3];
arr[0] = 13;
arr[1] = 10;
arr[2] = 16;
```
Peer instruction

• Suppose we allocate this array:
  ```java
  int[] arr = new int[7];
  ```
and Java decided to put the first byte of the array (first byte of arr[0]) at address 0x04

What is the address of arr[5]?

a. 0x04  
b. 0x14  
c. 0x09  
d. 0x54  
e. 0x05  
f. 0x18  

address = base address + index * element_size
## Size of data types (in bytes)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>size in 32-bit architecture</th>
<th>size in 64-bit architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float (later!)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>reference (stores a memory address)</td>
<td><strong>4</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

slide inspired by UW CSE351
Administrivia:

• All office hours/Debug Your Brain scheduled: see syllabus
• Waitlist is at 23; checking to see if you can enroll in the alternative ECE 3350 as backup (still has ~20 spots)
• course policy survey
  • so far: 2 slip days per assignment is clear winner
  • so let’s start using that for HW 1 (although we suggest saving them for later homeworks)
Aside: naming powers of two

• One of our favorite formulas: how many unique things can you represent with N bits? \(2^N\) things

• Naming conventions

<table>
<thead>
<tr>
<th>N = ?</th>
<th>name</th>
<th>Close (but not equal!) to power of 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Kibi (Ki)</td>
<td>(10^3) or Kilo (K)</td>
</tr>
<tr>
<td>20</td>
<td>Mebi (Mi)</td>
<td>(10^6) or Mega (M)</td>
</tr>
<tr>
<td>30</td>
<td>Gibi (Gi)</td>
<td>(10^9) or Giga (G)</td>
</tr>
<tr>
<td>40</td>
<td>Tebi (Ti)</td>
<td>(10^{12}) or Tera (T)</td>
</tr>
<tr>
<td>50</td>
<td>Pebi (Pi)</td>
<td>(10^{15}) or Peta (P)</td>
</tr>
<tr>
<td>60</td>
<td>Exbi (Ei)</td>
<td>(10^{18}) or Exa (E)</td>
</tr>
</tbody>
</table>

Name these numbers of bytes:

\(2^{11}\) bytes = _______

\(2^{32}\) bytes = _______

\(2^{59}\) bytes = _______
Where we are going (today)

Bits store whatever you want, including integers. How do we manipulate sequences of bits?

10110000

Q: How do we arrange bits in the memory of the computer? (why do we care? we want the computer to store many individual numbers)
A: bytes and words

Q: How do we name or refer to all those individual numbers in memory?
A: addresses and pointers

DONE (for now)

DONE (for now)
Instruction memory can do bitwise operations (and other stuff) Execution engine Data memory
Op 1. Changing the number of bits

• Often we need to change the number of bits we are using to store a number (why?)

• $12_{10}$ using 4 bits is $1100_2$
• If we store it in 8 bits: $00001100_2$

• $-5_{10}$ using 4 bits is: $1011$
• If we store it in 8 bits: _______________ $2$

We call this operation **sign extension**: copy the leftmost bit in 4-bit number to the new 4 leftmost bits in the 8-bit number
Op 2. Bitwise operations

| Logical AND (&)   | Logical OR (||) |
|-------------------|-----------------|
| 0011              | 0011            |
| 1010              | 1010            |
|                   |                 |
| 0010              | 1011            |
|                   |                 |
| 0011              | 0110            |
| 1010              |                 |
|                   | 0110            |
|                   | 1001            |
Representing sets with bits

• 4-bit vector represents subsets of \{3,2,1,0\}
  • 1 means in the set, 0 means not in the set

• A is the set \{1,0\}, encoded as 0011
• B is the set \{2,0\}, encoded as 0101

• Set operations using bitwise operators
  • A&B Intersection 0001 (1 iff both are 1)
  • A|B Union 0111 (1 iff at least one 1)
  • A^B symmetric difference 0110 (1 iff exactly one 1)
  • ~B complement 1010 (1 iff 0)
Op 3. Shift

• shift: move bits left or right

• Left shift: 6 << 2

• Right shift: 6 >> 1

(diagrams show that I am storing the integers using 5 bits)
Peer instruction

What is the integer result from evaluating this expression? (assume integers are 32 bits)

\((7 << 2) \& 15\)
Let’s play a card game

- Come up with a binary encoding for a 52-card deck

Operations on a pair of cards

We want the following operations to be easy to implement

- Compare two cards, which is higher value?
- Compare two cards, are they the same suit?
Summary and what’s next

TODAY can do bitwise operations (and other stuff)

NEXT Stores programs!

TODAY organizes its bits as **bytes** and **words**
What to do next

• HW1/Quiz1 due Wednesday
• Readings up on MIPS
• Introduce yourself at office hours/debug your brain on Tues, Wed, Th
• HW2 will be on the next topic of MIPS programming and is out immediately after HW1
  • Install MARS on the resources page of the website if you have extra time, you’ll use it in HW2
Where we are going

- Compiler
  - translating source code (C or Java)
  - Programs to assembly language
  - And linking your code to
  - Library code

- Instruction set architecture (e.g., MIPS)
  - How the software talks
  - To the hardware

- Memory system

- Processor
  - How a processor runs MIPS
  - Programs!

- I/O system
  - How switches (1 or 0) can be used
  - to build Interesting functions:
  - from integer arithmetic to
  - programmable computers

- Datapath & Control

- Digital logic
Where we are going

Instruction memory

Execution engine

Data memory

Learn what gets stored in here
Organize bytes into machine words

(note address in hex)

32-bit words

64-bit words
We have a hypothetical word-addressed machine with 32-bit addresses. Assume array $x$ starts at address $0x50$. 

```
int[] x = new int[4];
```

What is the address of $x[3]$?

a) $0x50$
b) $0x53$
c) $0x5C$
d) $0x60$
e) $0x54$
Review of bitwise operations

<table>
<thead>
<tr>
<th>Logical AND (&amp;)</th>
<th>Logical OR (|)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>____</td>
<td>____</td>
</tr>
<tr>
<td>0010</td>
<td>1011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XOR (^)</th>
<th>negate (~)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0110</td>
</tr>
<tr>
<td>1010</td>
<td>0110</td>
</tr>
<tr>
<td>____</td>
<td>____</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
</tr>
</tbody>
</table>
Shift

- shift: move bits left or right
- Left shift: 6 << 2
- Right shift: 6 >> 1
Where we are going

Instruction set architecture (e.g., MIPS)

Compiler
translating source code (C or Java)
Programs to assembly language
And linking your code to
Library code

Instruction set architecture (e.g., MIPS)

Memory system

Processor

I/O system

Datapath & Control

How the software talks To the hardware

How a processor runs MIPS Programs!

How switches (1 or 0) can be used to build Interesting functions:
from integer arithmetic to programmable computers

Digital logic
Representations of a program

int x = arr[1];
arr[2] = x + 10;

Assembler

lw $t0, 4($r0)
addi $t0, $t0, 10
sw $t0, 8($r0)

assembly program as text
(CS2630 student readable)

assembly program as binary
(machine readable)
Languages

• High level language (HLL) programs are **machine-independent**
  • language is convenient to use
  • language is has powerful features

• Assembly language programs are **specific to an architecture**
  • the “native language” of the processor
  • language is bare-bones: lacks most of the features of HLLs
Brief history of compatibility

pre IBM 360

my_program.asm
version 1

Processor model 1
slow processor, small
memory

my_program.asm
version 2

Processor model 2
mediocre processor, medium memory

my_program.asm
version 3

Processor model 3
fast processor, large memory

post IBM 360

my_program.asm

Processor 1
slow processor, small memory

Processor 2
mediocre processor, medium memory

Processor 3
fast processor, large memory
Peer instruction

I wrote my program in C (a high level language) and compiled it to run on a MIPS machine. What should I do to get that program to run on an x86 machine?

a) rewrite my C program to be compatible with x86
b) rewrite my program in x86 assembly language
c) recompile my C program for x86
d) reassemble my program using the x86 assembler
Languages

• High level language: usually has variables, objects, and arrays

• Assembly language: usually just has registers and memory
Microprocessor Transistor Counts 1971-2011

https://library.law.uiowa.edu/history-library
memory

registers
The main idea

• 4 categories of *instructions*
  • perform an operation on two registers and store result in a register
  • perform an operation on one register and a constant and store the result in a register
  • move a value between a register and memory
  • determine which instruction to execute next
Register transfer language

• 4 categories of *instructions*
  • perform an operation on two registers and store result in a register
  • perform an operation on one register and a constant and store the result in a register
  • move a value between a register and memory
  • determine which instruction to execute next

(later!)
Example assembly program

Peer instruction: Write an assembly program using a sequence of instructions (use register transfer language)
Introducing MIPS

```
lw $t0, 4($0)  # r0 <- M[0x4]
lw $t1, 8($0)  # r1 <- M[0x8]
add $t2, $t0, $t1  # r2 <- r0 + r1
lw $t0, 12($0)  # r0 <- M[0xC]
sub $t2, $t2, $t0  # r2 <- r2 - r0
sw $t2, 0($0)   # M[0x0] <- r2
```
# MIPS registers (32 of them)

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Conventional Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>hard-wired to 0</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$2 - $3</td>
<td>$v0, $v1</td>
<td>return values from functions</td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$a0 - $a3</td>
<td>arguments</td>
</tr>
<tr>
<td>$8 - $15</td>
<td>$t0 - $t7</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>$s0 - $s7</td>
<td>saved registers</td>
</tr>
<tr>
<td>$24 - $25</td>
<td>$t8 - $t9</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$26 - $27</td>
<td>$k0 - $k1</td>
<td>reserved for OS</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>stack Pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>frame Pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>return Address</td>
</tr>
</tbody>
</table>
MIPS instruction format: **R-type**

```
add  $t2, $t0, $t1
   dst  src  src

sub  $t6, $t7, $t2
   dst  src  src

or   $t0, $t3, $t4
   dst  src  src
```
MIPS instruction format: I-type

lw $t0, 4($0)

\[\text{dst} \quad \text{Imm} \quad \text{src}\]

“load word”

addi $t1, $t5, 100

\[\text{dst} \quad \text{src} \quad \text{Imm}\]

“add immediate”

sw $t2, 0($0)

\[\text{dst} \quad \text{Imm} \quad \text{src}\]

“store word”

I is for *immediate* which in MIPS means a constant
load and store

\[
\text{lw } \$t0, \ 4(\$0) \\
\text{dst} \quad \text{Imm} \quad \text{src}
\]

address = base + offset
= src + Imm
= 0 + 4 = 4

\[
\text{sw } \$t2, \ 0(\$0) \\
\text{dst} \quad \text{Imm} \quad \text{src}
\]

address = base + offset
= src + Imm
= 0 + 0 = 0

remember, register $0$ (or $\$zero$) is hard-wired to $0$
Logical operations

• R-type
  • sll $t0, $t1, 2      # t0 <- t1 << 12  (shift left logical)
  • srl $t0, $t1, 3     # t0 <- t1 >> 4   (shift right logical)
  • sra $t0, $t1, 3     # t0 <- t1 >>> 4  (shift right arithmetic)
  • and $t0, $t1, $t2   # t00 <- t1 & t2  (AND)
  • or $t0, $t1, 2      # t0 <- t1 | t2   (OR)

• I-type
  • andi $t0, $t1, 22   # t0 <- t1 & 22
  • ori  $t0, $t1, 0xFF # t0 <- t1 | 0xFF

yes, sll and srl are R-type; we’ll soon see why (hint: what is the largest shift value needed?)
Translate the following code into MIPS.

// k is an int[]. Assume the first byte of x is already
// stored at address 0x10
k[0] = k[1] + 1;
What are we missing?
Making decisions

if (z == 0) then a = x + y
else a = z

bne “branch (on) not equal”
j “(unconditional) jump”

```
bne $t0,$zero,there_plz
add $t1,$t2,$t3
j finish_plz
there_plz: or $t1,$t0,$zero
finish_plz: sw $t1,0($t7)
```
A Case against the GO TO Statement.

by Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code).

article AKA “Go-to statement considered harmful”
Branch/jump instructions

```

bne  $t0,$zero,there_plz
    src    src    label

beq  $t1,$t3,launch
    src    src    label

j    over_the_moon
    label

jr   $t5
    src
```

“branch on not equal”

“branch on equal”

“jump”

“jump register”
Peer instruction

What value is stored in $t0 when the program finishes?

```
ori  $t1,$zero,3
addi $t2,$zero,1
yum: addiu $t1,$t1,-1
sll  $t2,$t2,1
bne  $t1,$zero,yum
nop # no-op, this instruction does nothing
addiu $t2,$t2,1
end: or  $t0,$zero,$t2
```

(numeric response)
John Von Neumann Documentary

https://youtu.be/VTS9O0CoVng?t=34m12s
Stored program concept

- Instruction memory
- Execution engine
- Data memory

Program represented as bits and stored here