CS/COE0447: Computer Organization and Assembly Language

Bruce Childers

MW 3:00-4:15
MW 4:30-5:45

Spring 2016

Dept. of Computer Science
University of Pittsburgh

Course Details

Web site: http://www.cs.pitt.edu/~childers/CS0447
** Look here! Syllabus & News! **

Book: *Computer Org. and Design* by Patterson and Hennessy, 5th Ed., M.K.
Software: MARS (MIPS simulator) and Logisim (Logic simulator) **FREE!**

A tale of three topics (1/3 semester each): MIPS, logic, processor design

Recitation: Required. Best 10 count toward grade. Weekly 5 minute quiz.
** Attend only your registered recitation!! **
Projects: One significant MIPS, one significant logic ** New & improved! **
Exams: 2 midterms, 1 final exam (date is fixed – see web site)
Grading: Exams 15%, 15%, 20%; Projects 15%, 15%; Lab 20%, Lab quiz 0%

Late assignments: 20% penalty each day late w/o pre-approved excuse
Make-up exams: Must make prior arrangements! BEFORE the exam.
Regrading: Sure! Quibbling over a few points isn't worth it. Write up explanation.

Computer systems

Three general classes of "computer"

"Desktop computers"

- Examples include PC, Mac, Chrome, Linux…
- Notebooks, netbooks, tablets (smart phones), …
- Interact with a user – applications
- Handful of central processing units (4-12?), gigabytes (10^9 bytes) memory, few terabytes (10^{12} bytes) of disk
- 35 gigaflops (35×10^9 "floating-point math calculations" per second for Intel Ivy Bridge)

NOT a trash can!

Trash can

Computer systems

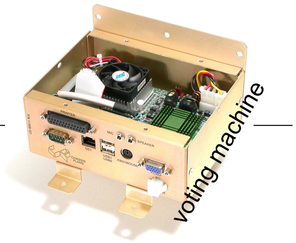
Three general classes of "computer"

"Desktop computers"

"Servers"

- Web servers, Computational servers, Supercomputers
- Interact with other computers to "solve a problem" or "provide services"
- Dozens to thousands of CPUs (Tianhe-2: 3,120,000 CPUs, 33.9 petaflops, or $33.9\times\boldsymbol{10^{15}}$ calculations per second vs. 35×10^9 per second for PC)
- Gigabytes to terabytes memory (Sequoia: 1,024,000GB [1.0 petabyte!])
- Petabytes (10^{15} bytes) of storage
- Connected (network) to work together
- Power hungry but efficient (Tianhe-2: 17.8 MW vs. Three Mile Island ~800 MW output. Data centers: 1.7% to 2.2% of total electricity in US.)

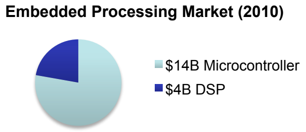
Computer systems



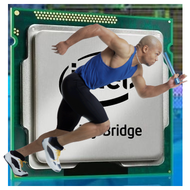
Three general classes of “computer”

“Desktop computers”
“Servers”
“Embedded computers”

- Hidden inside something not computer
- Applications that run on these computers interact with the “real world”
- Multiple different processors for different functions
- Kilobytes (10^3 bytes) to gigabytes of memory
- Kilobytes to gigabytes of storage
- Slow speed to fast speed
- Widest range of design!



Computer systems: Commonality



Programmable: Software programs “run” on the hardware

Components

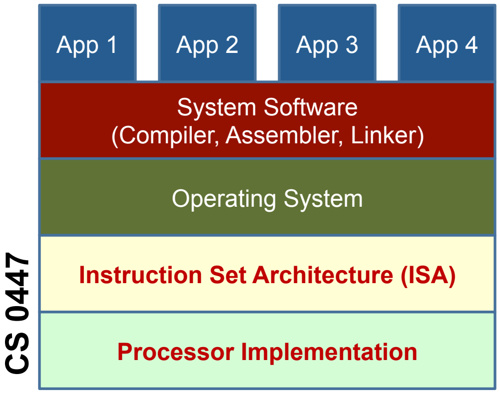
- ***Central processing unit (CPU)***: Does the computation
  - A.k.a., “the processor”, “the core”
- **Main memory**: Temporarily holds results (volatile)
- **Storage**: Long term storage (permanence) for large quantities
- **Input/Output**: Interaction (human, physical world or machine)

Metrics

- **Speed**: How fast computation is done. Faster is not always necessarily better. Usually some constraint/goal on speed.
- **Energy/Power**: A BIG concern today! Battery. Electricity cost and delivery to data center.

Layers or views

Our view of a computer system in this course is centered around the interface between the lowest level in software and the hardware



Where do we start????

There's a lot to cover in CS 447...

- Software-hardware interface: "Instruction set architecture"
- MIPS assembly language programming and concepts
- Number representation and binary arithmetic
- Logic design (AND, OR, NOT)
- The building blocks of computation in the CPU
- Building your very own CPU

Binary numbers are fundamental!

- Everything is really just an operation on binary numbers
- The CPU "understands" only binary numbers
- So, we need to first understand some basics
- Gives the entire class a common basis for discussion

Numbers

You encounter a form (taxes, graduation, etc.) and you see a field labeled **year** like so:

What is the **smallest** year you could put in the box?

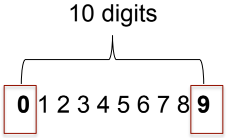
0	0	0	0

What is the **largest** year you could put in the box?

9	9	9	9

Why?

Well we simply can pick all of the smallest digits and all of the largest digits:



Range

How many total values can we put in the box?

Range = High - Low +1
Range = 9999 - 0000 + 1
Range = 10,000

Let's write that a different way:

$$\mathbf{10^4}$$

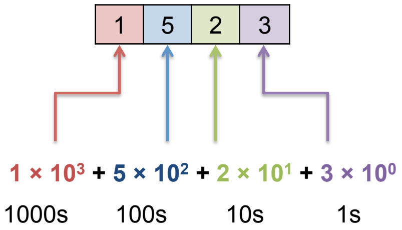
We see a **10** (the number of digits) and a **4** (the number of boxes). Is this a coincidence?

No. It's a property of how we write numbers.

Positional Number Systems

This is what we learned in grade school.

There is a ones' place and a tens' place and a hundreds' place … etc.



1	5	2	3

$$1 \times 10^3 + 5 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

1000s 100s 10s 1s

We call this a **positional number system** because the position of each digit tells us the magnitude of the value.

Each position is a higher *exponent* on a **base**. In daily life, we typically use base 10, also known as **decimal**.

Do Other Bases Make Sense?

Can we still have a positional number system with a base other than 10?

Yes. Any number can be a base, but for our purposes some are more useful than others.

Base 2 – Binary
Base 8 – Octal
Base 16 – Hexadecimal

Base 2: Binary

When we have a base N, the allowable digits are [0,N-1].
So for base 2, we only use 0 and 1.

A binary digit is known as a **bit** (a contraction of **bi**nary dig**it**).

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

Decimal	Binary
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Binary addition

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0, carry = 1

Examples

101 + 0 = 101
111 + 10 = 1001
1011 + 11 = 1110

Bits, Nibbles, and Bytes

Each 0 or 1 in a binary "string" is a **bit**. It is designated with a lowercase b.

- Binary strings of length 4 are called a **nibble** (or nybble).
- Binary strings of length 8 are called a **byte**. Designated with a B.
- Bytes aggregated into groups called **word**s. Word size can vary depending on the computer architecture. Often a word will be 16, 32 or 64 bits (2, 4, 8 bytes).

Oftentimes, the **byte** is the element that can hold one character of text in English.

The **byte** is usually the smallest addressable memory element on a machine.

The size of a byte being 8 bits was not common until the 1970s and the term *octet* was sometimes used to avoid confusion.

Binary to Decimal Conversion

What is decimal value for 1001001101b?

1	0	0	1	0	0	1	1	0	1
512	256	128	64	32	16	8	4	2	1
2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Take each position that has a 1 in it and add up the corresponding powers of two:

512 + 64 + 8 + 4 + 1 = 589

Quick Sanity Check: If the ones' place (2^0) is 1, then the number must be odd!

Decimal to Binary Conversion

For a decimal input called **value**:

1. **Start**: Find the biggest power of 2 smaller than **value**
2. if value/(that power) == 1
    a) Output a "1"
    b) Subtract that power from **value** and store back in **value**
3. Else
    a) Output a "0"
4. Move to the next smaller power of 2
5. Go to 2 while we haven't done the one's place

Decimal to Binary Example

Input value: 75

Start: Largest power of 2 less than 75? 64

Divide*	**New value**	**Next power**	**Output**	
75 / 64 = 1	75-64=11	32	1	*64s*
11 / 32 = 0	11	16	0	*32s*
11 / 16 = 0	11	8	0	*16s*
11 / 8 = 1	11-8=3	4	1	*8s*
3 / 4 = 0	3	2	0	*4s*
3 / 2 = 1	3-2=1	1	1	*2s*
1 / 1 = 1	1-1=0	0 (done)	1	*1s*

* integer division

Result: 1001011

Check yourself: $2^6 + 2^3 + 2^1 + 2^0 = 64 + 8 + 2 + 1 = 75$ (it worked! yea!)

Base 8: Octal

Bit strings can be ***very*** long and sometimes we wish to compactly represent, while easily converting in and out of binary.

Octal is base 8.

The valid digits are then [0,7]

Every 3 bits can be represented with one octal digit.

Programming languages usually denote octal literals with a leading 0 prefix.

Base 16: Hexadecimal

More common is ***base 16, called hexadecimal*** or just “hex” for short.

Every sequence of 4 bits is represented with a single hexadecimal digit.
Thus, 32-bit numbers are compactly displayed in 8 hex digits.

Each digit ranges from [0, ***15***??]

Cannot use 2 digits for one as that will destroy positional number.
We need new “digits” for 10, 11, 12, 13, 14, and 15.

Solution? ***Use letters: A, B, C, D, E, F.***

Range is [0,F] i.e., 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Counting in the Bases

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
0	0000	0	0	8	1000	10	8
1	0001	1	1	9	1001	11	9
2	0010	2	2	10	1010	12	A
3	0011	3	3	11	1011	13	B
4	0100	4	4	12	1100	14	C
5	0101	5	5	13	1101	15	D
6	0110	6	6	14	1110	16	E
7	0111	7	7	15	1111	17	F

Can you convert 0x7E1 to binary?
And then to decimal?

Um, so why binary?

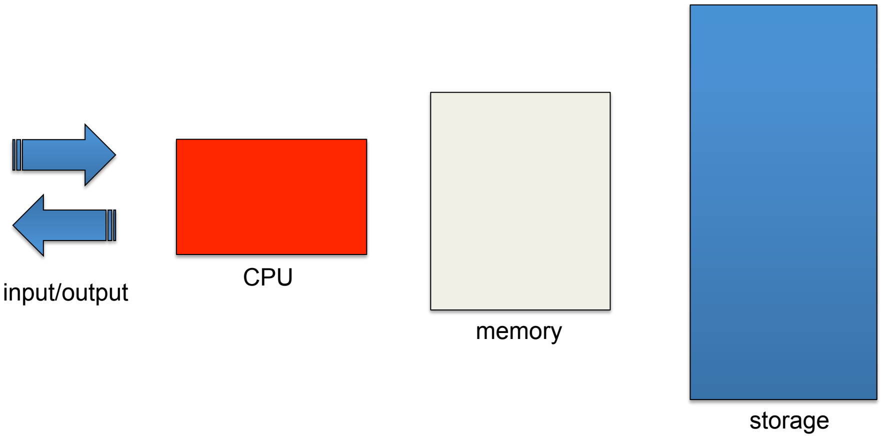
Digital computers are built around “switches” (transistors)
  - Switch has “on” and “off” state

Really, it’s about Boolean logic.
  - Your new best friends: AND, OR, NOT
  - Computation (circuit) defined as functions of these operations
  - Boolean logic has two values: **True**, **False**

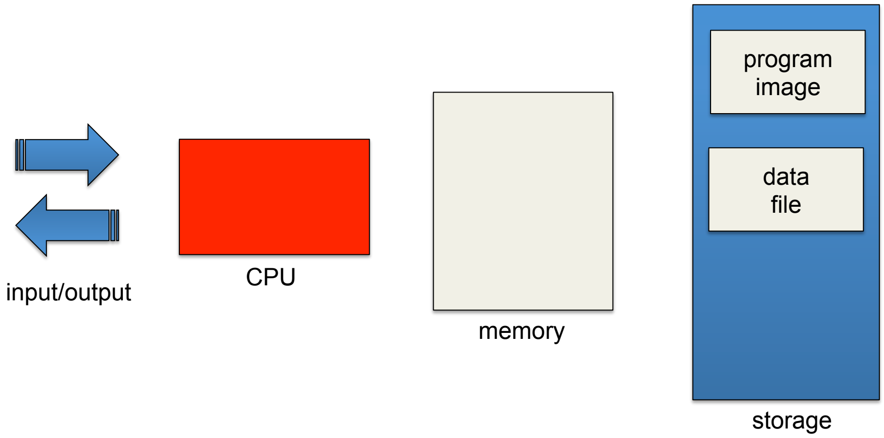
Hmm. A switch is “on” or “off”. Two values in logic. T, F. Binary is 0, 1.
  - **Light bulb moment!** Since Boolean logic has two values, processing is built around logic functions, then naturally, we use binary…

Note: It’s quite possible to build a processor in other bases. Analog computing!

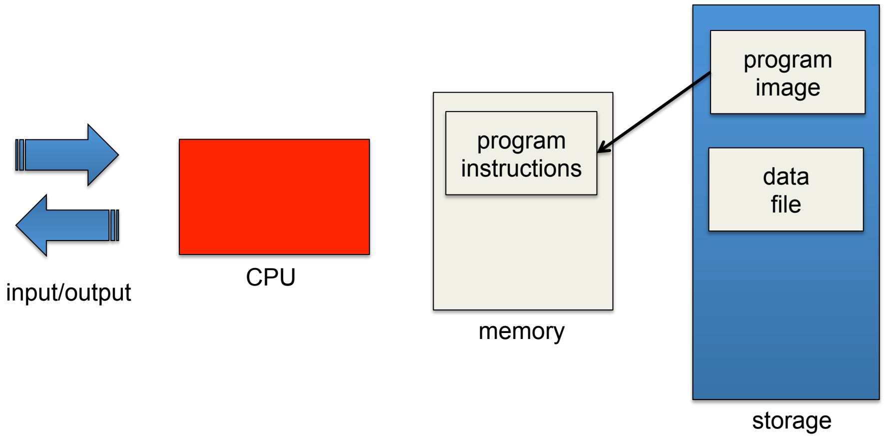
Components of a Computer



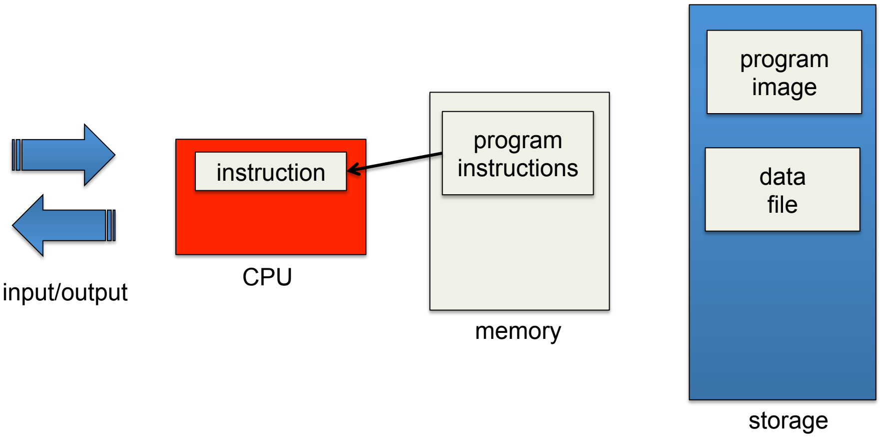
Running a Program



Running a Program



Running a Program



input/output

instruction

CPU

program instructions

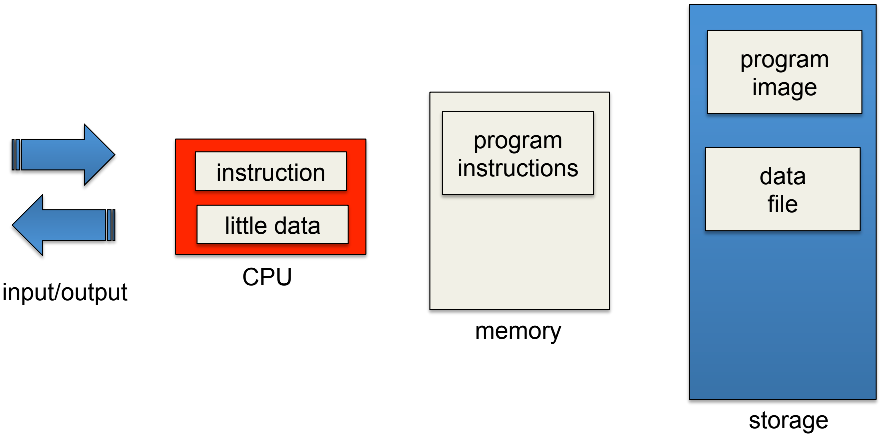
memory

program image

data file

storage

Running a Program



input/output

instruction

little data

CPU

program instructions

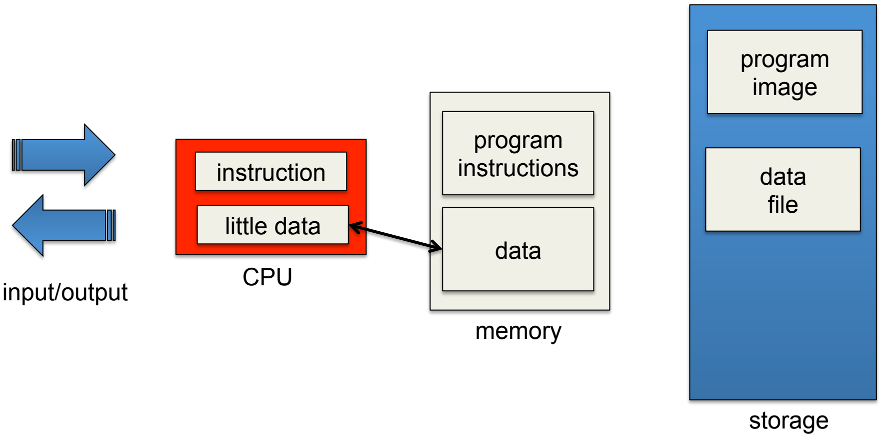
memory

program image

data file

storage

Running a Program



input/output

instruction

little data

CPU

program instructions

data

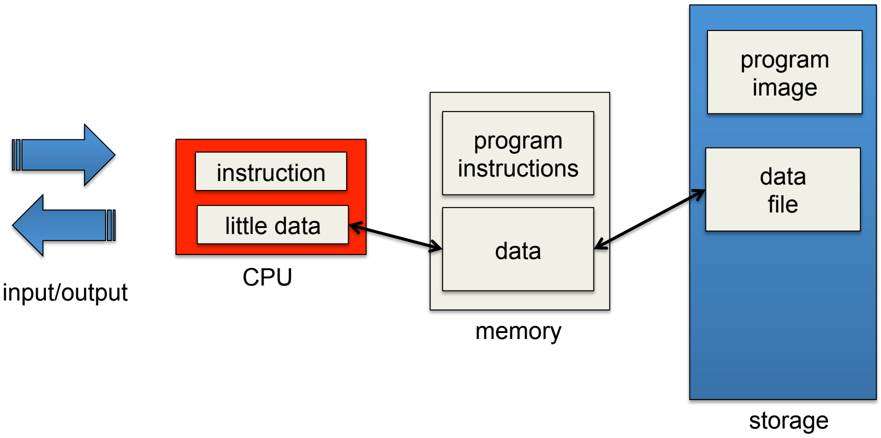
memory

program image

data file

storage

Running a Program



input/output

instruction

little data

CPU

program instructions

data

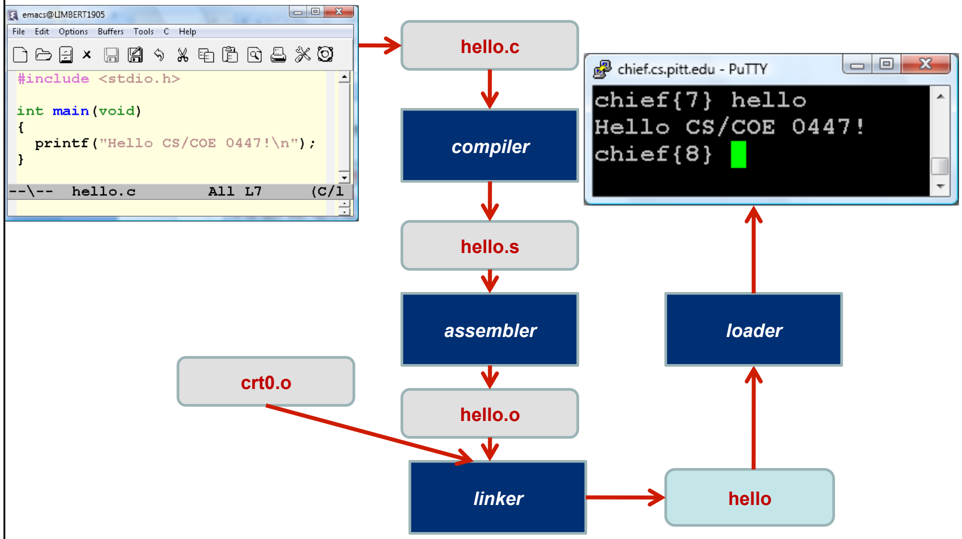
memory

program image

data file

storage

A Day in the Life of a C (Java) Program



```
#include <stdio.h>

int main(void)
{
  printf("Hello CS/COE 0447!\n");
}
--\--  hello.c         All L7      (C/1
```

```
chief{7} hello
Hello CS/COE 0447!
chief{8}
```

Assembly Language, Machine Code

High-level programming languages are a convenience

- CPU ***does not*** "understand" the high-level language!

CPU understands "binary numbers"

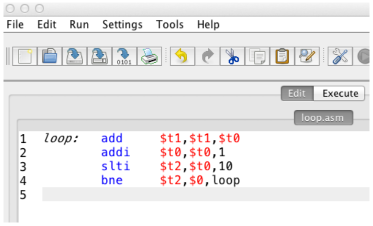
- Binary number represents a ***command***
- A command makes CPU take some action (e.g., addition)
- Commands known as "***machine instructions***" (code)

Who wants to program in binary numbers???

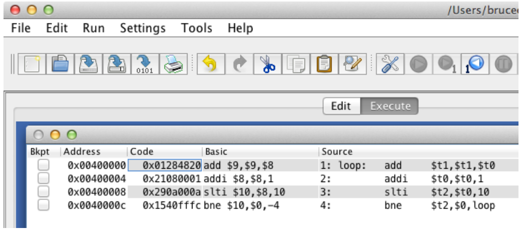
- ***Assembly language*** is convenience
- Programming in machine instructions

Assembling, Loading, Running

- Assembly language program is *assembled*
  - ***Assembler*** is tool to create *machine code from assembly language*
- Assembled program placed in main memory
  - ***Loader*** is tool to put the machine instructions into memory
  - Loader is automatically used when you run the program
- CPU gets access to machine instructions in memory
- CPU does the command for each instruction

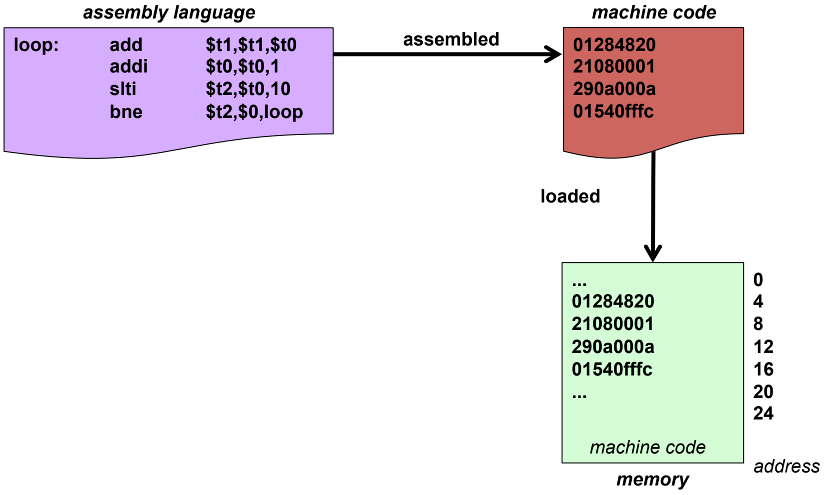


```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x01284820	add $9,$9,$8	1: loop: add $t1,$t1,$t0
☐	0x00400004	0x21080001	addi $8,$8,1	2: addi $t0,$t0,1
☐	0x00400008	0x290a000a	slti $10,$8,10	3: slti $t2,$t0,10
☐	0x0040000c	0x1540fffc	bne $10,$0,-4	4: bne $t2,$0,loop

Assembling, Loading, Running



assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

assembled

machine code

01284820
21080001
290a000a
01540fffc

loaded

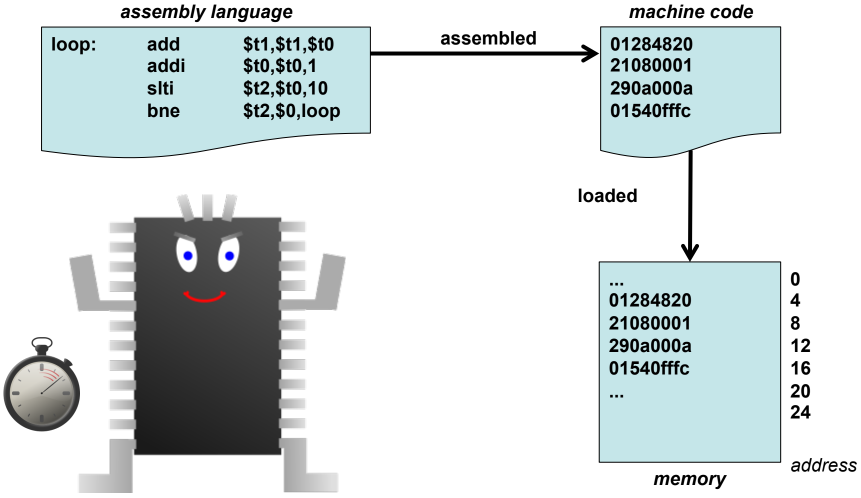
...	**0**
01284820	**4**
21080001	**8**
290a000a	**12**
01540fffc	**16**
...	**20**
	24

machine code

memory

address

Assembling, Loading, Running



assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

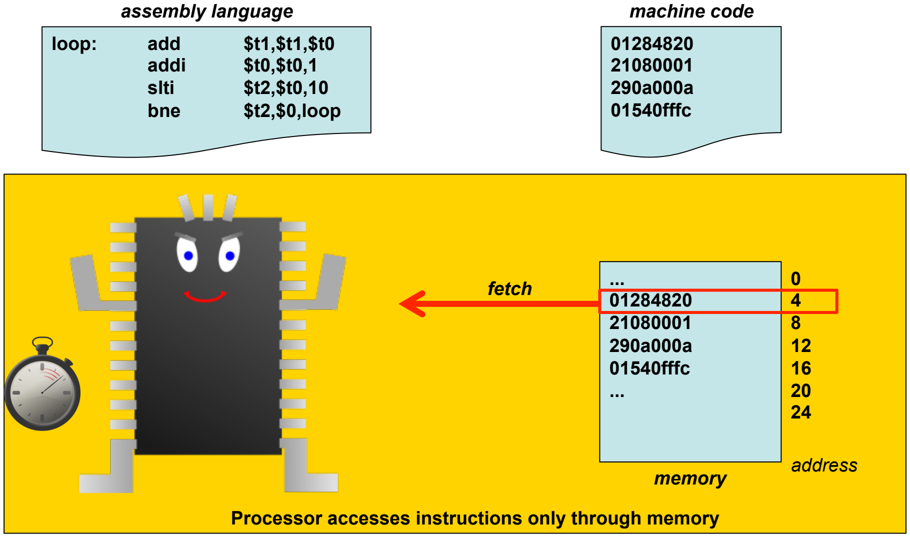
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Assembling, Loading, Running



assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

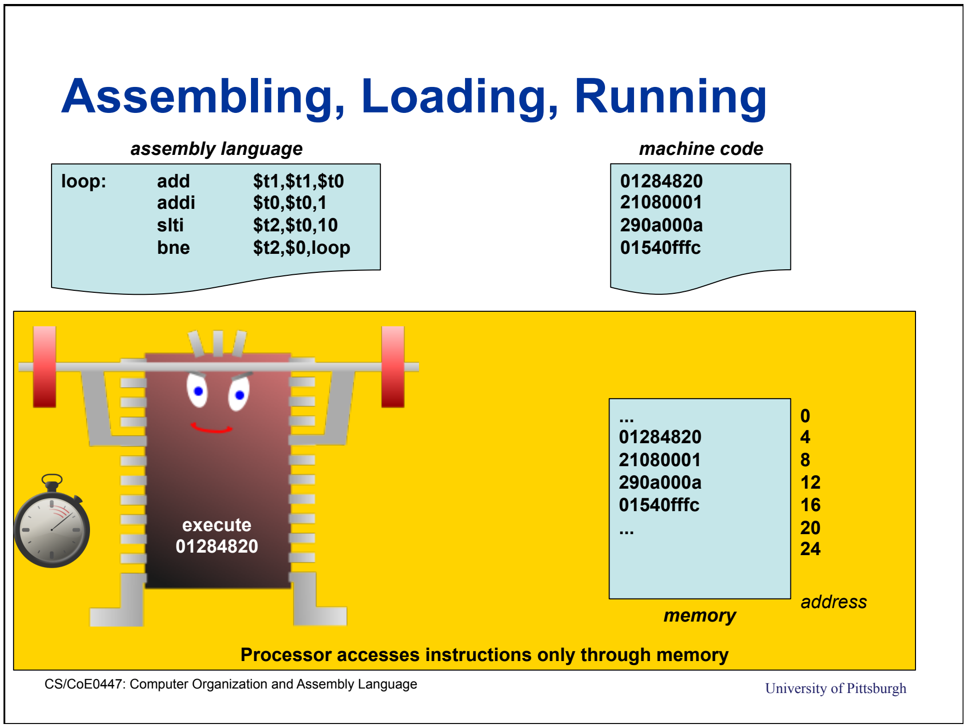
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

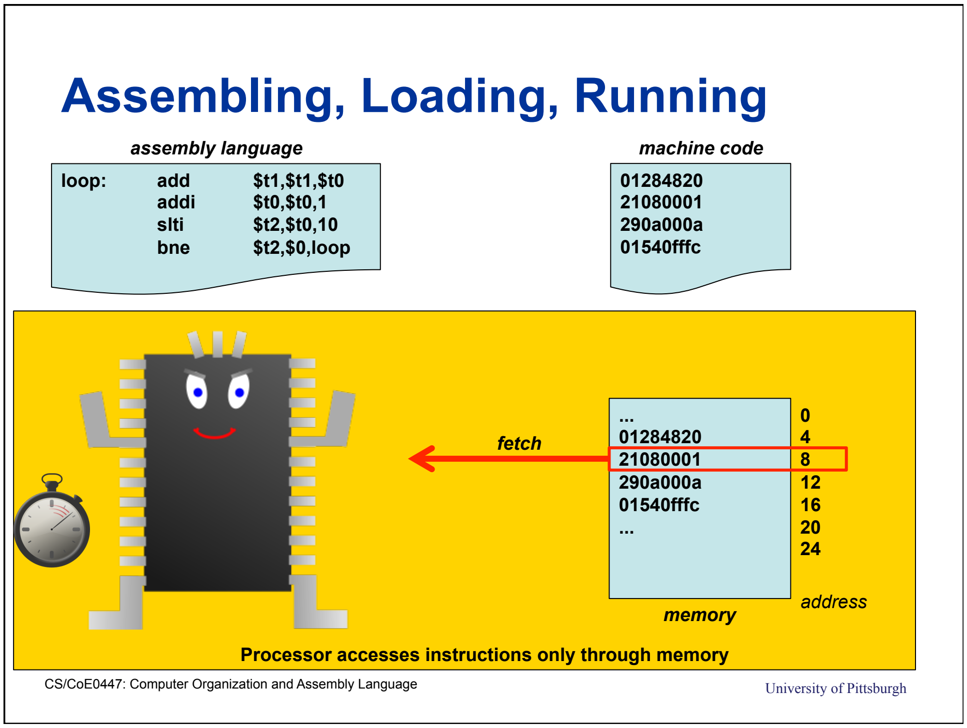
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

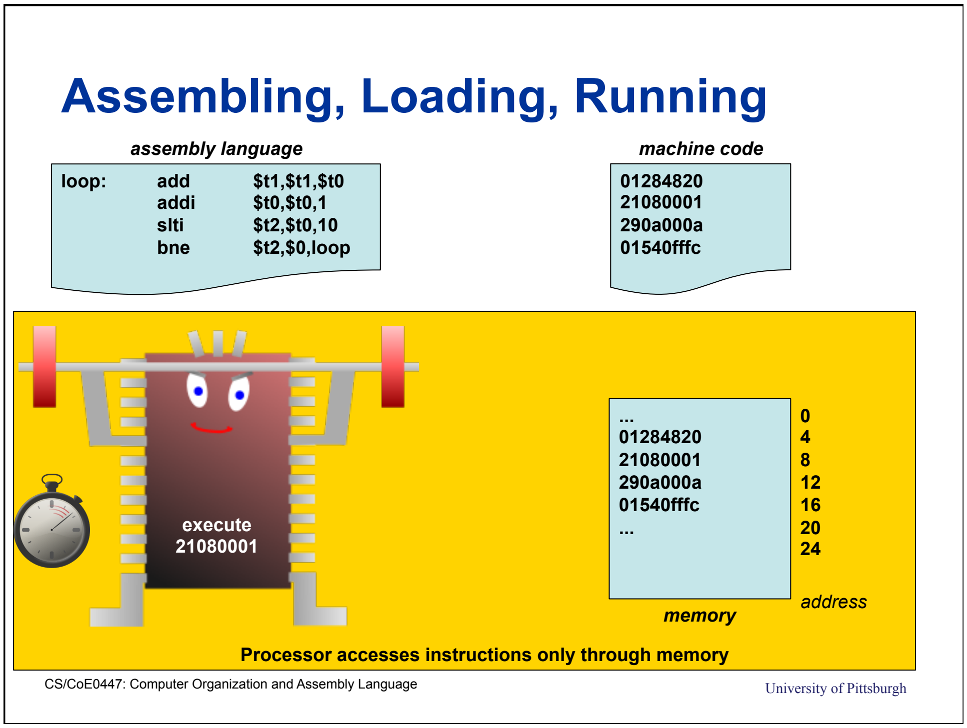
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

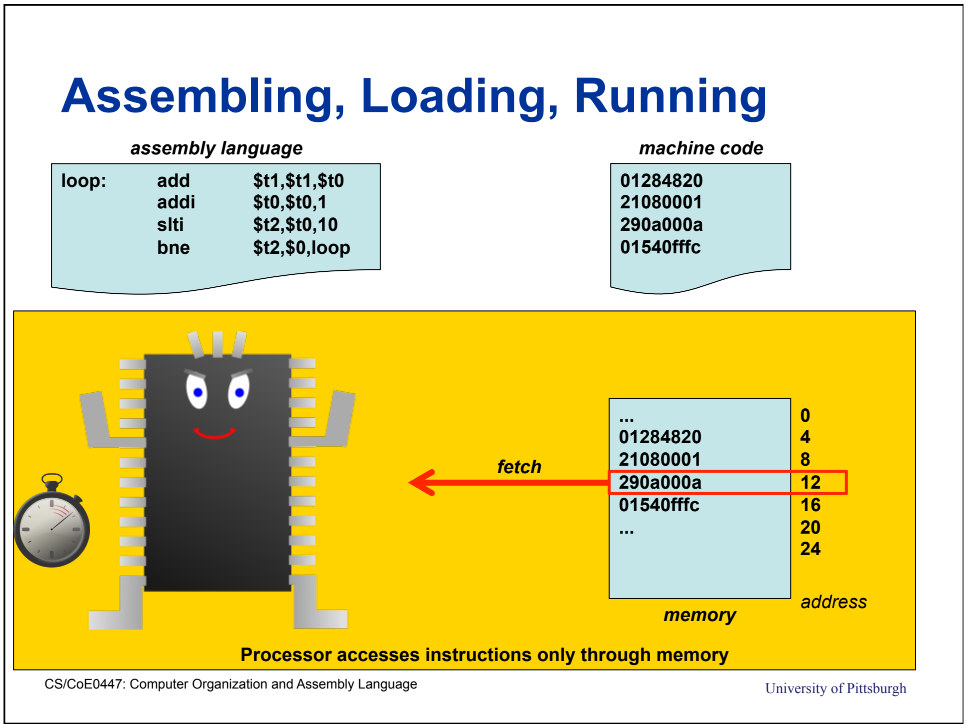
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

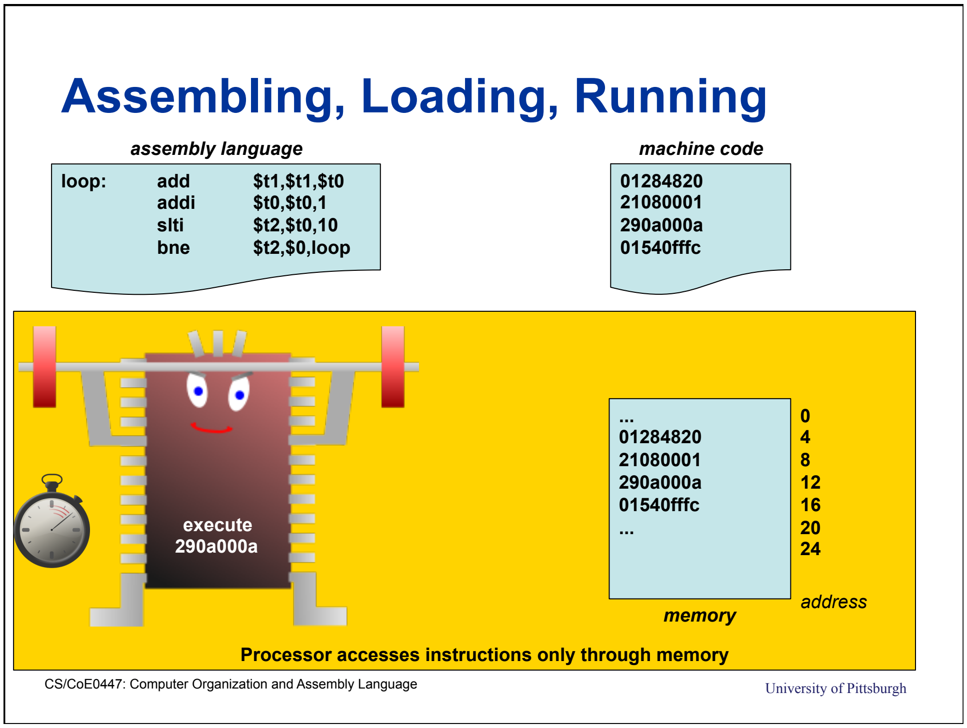
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

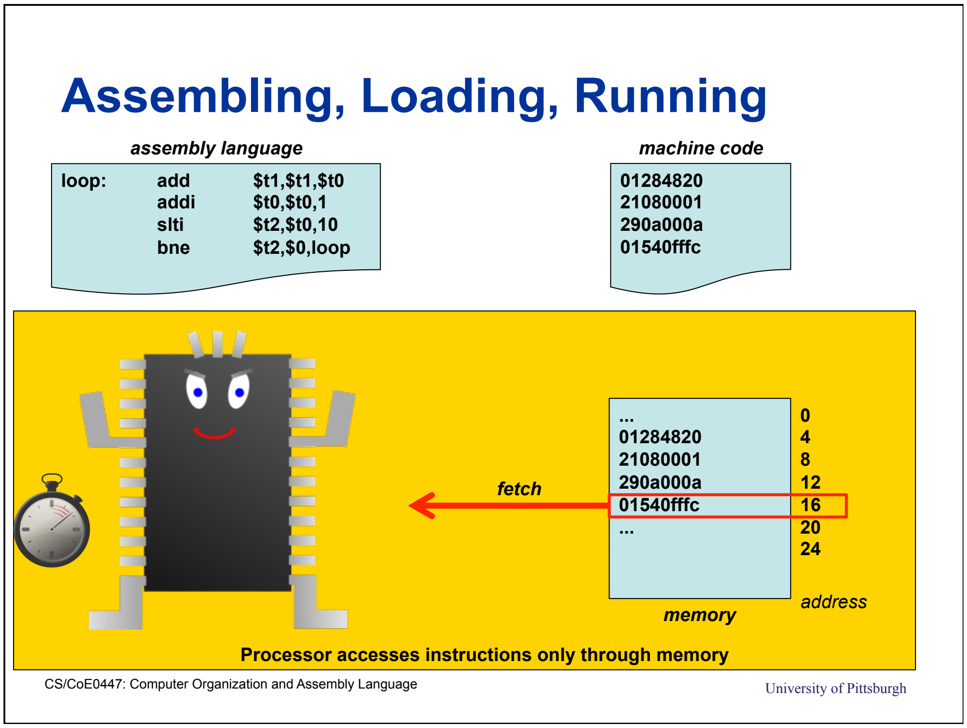
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

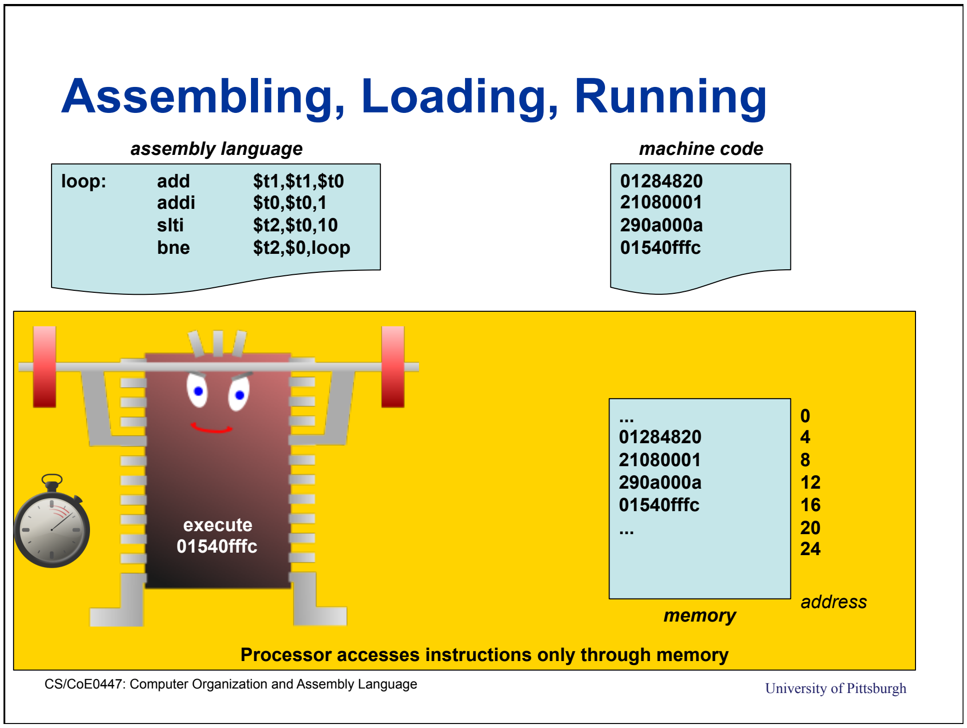
machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

memory *address*

Processor accesses instructions only through memory



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

machine code

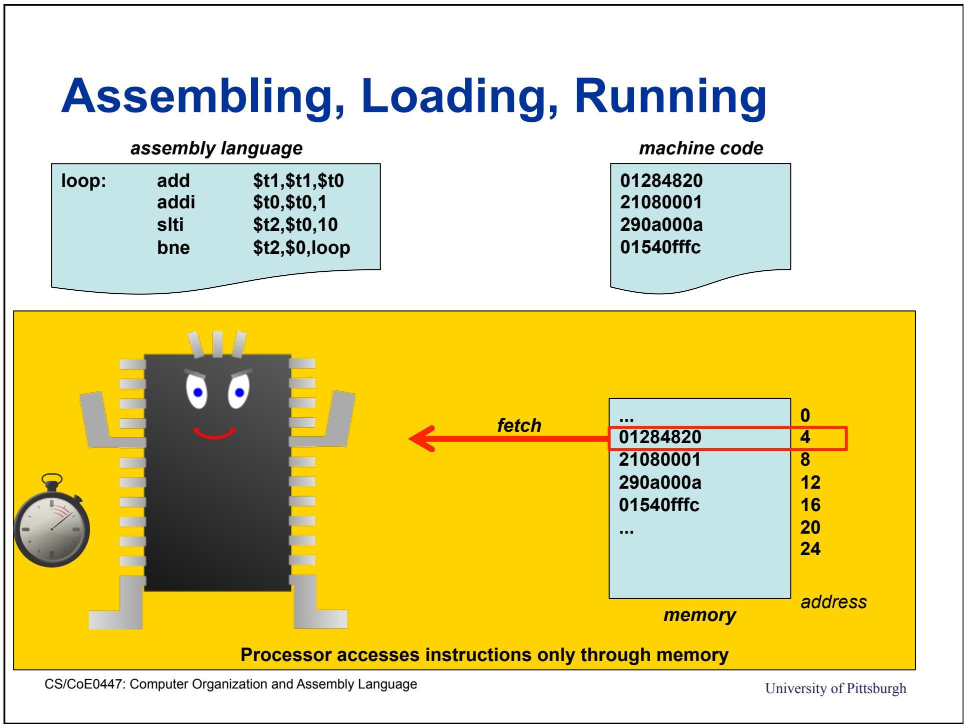
```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24
memory	*address*

Processor accesses instructions only through memory

CS/CoE0447: Computer Organization and Assembly Language

University of Pittsburgh



Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

machine code

```
01284820
21080001
290a000a
01540fffc
```

...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24
memory	*address*

Processor accesses instructions only through memory

CS/CoE0447: Computer Organization and Assembly Language

University of Pittsburgh

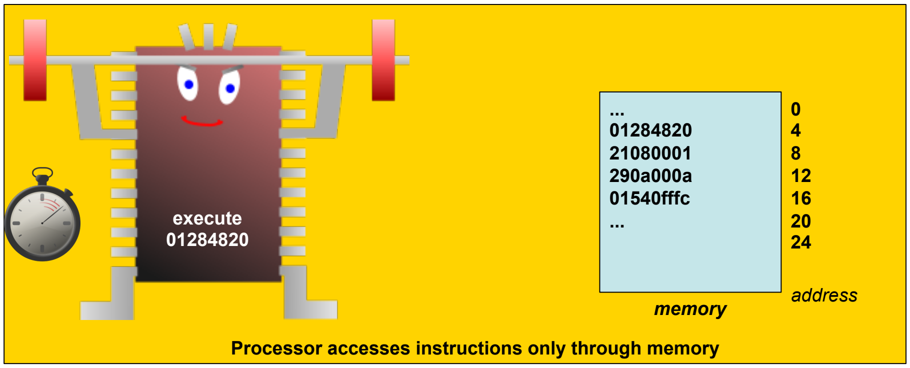
Assembling, Loading, Running

assembly language

```
loop:   add     $t1,$t1,$t0
        addi    $t0,$t0,1
        slti    $t2,$t0,10
        bne     $t2,$0,loop
```

machine code

```
01284820
21080001
290a000a
01540fffc
```

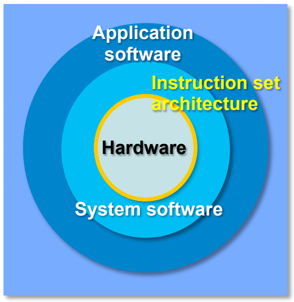


memory	*address*
...	0
01284820	4
21080001	8
290a000a	12
01540fffc	16
...	20
	24

Processor accesses instructions only through memory

Instruction set architecture

- ISA is a programmer ***interface*** to the hardware
- Programmer's Reference Manual (PRM) discloses ISA
- Different ***implementations*** of the same ISA
- "Architecture" == ISA, "Microarchitecture" == Implementation



- You are a system software programmer
- Components of ISA in PRM
  - Data types the processor supports
  - **Registers and their usage**
  - **Instructions and their definitions**
  - **Processor modes**
  - Exception mechanism
  - (Compatibility issues)

Register

- It's ***storage in your processor*** that you can directly address and access in an instruction
- If your processor is 32-bit, your registers are (usually) 32 bits wide
- Depending on the processor, there can be many registers or only a few of those
  - Registers were a scare resource – they occupy chip space
  - Today we can put many registers; the concern is the access time and the power consumption

Instruction

- Unit of program execution; program consists of instructions
- **Describes an operation that the processor understands how to perform**
- The amount of work defined for an instruction is usually small
  - Add two numbers in registers (**add $t0,$t1,$t2**)
  - Compare two numbers in registers (**slt $t0,$t1,$t2**)
  - Make a jump in the program if the first number is smaller than the second number
- Complex instructions may ease your programming…
  - For example, "multiply two numbers from memory location A & B and iterate this 100 times or until you meet two zeros"
  - BUT, your processor implementation can become quite complex (slow!)

Processor modes

- "User mode"
  - Ordinary programs run in this mode
  - Most instructions can be executed in this mode (e.g., add, load)
  - Critical system resources are not directly accessed
  - What about other users' programs?
- "Privileged mode"
  - System software runs in this mode
  - Some instructions can be executed only in this mode
  - Critical system resources managed by the system software (i.e., OS)

Switching between modes

- When powered on, a processor will be in its privileged mode
- When the system boots up and becomes initialized, the system starts to execute user programs or interact with the user
- The processor switches back and forth between the modes when
  - There is an *exception*
    - E.g., Divide-by-zero, access something invalid
    - Program needs help from operating system
  - There is an *interrupt from input/output*
    - Clock interrupt (possibly causing another program to run)
    - Keyboard & mouse

Time to learn MIPS!