Instruction Set Architecture (2)

Prof. Usagi
Recap: von Neumann Architecture

By loading different programs into memory, your computer can perform different functions.
Recap: Instruction Set Architecture (ISA)

- The **contract** between the hardware and software
- Provide an **abstraction** of the underlying processor
  - Defines the set of operations that a computer/processor can execute
  - Defines the **memory** space that a program can use
- Programs are combinations of these instructions
- The hardware implements these instructions in any way it choose.
  - Directly in hardware circuit. e.g. CPU
  - Software virtual machine. e.g. VirtualPC
  - Simulator/Emulator. e.g. DeSmuME
  - Trained monkey with pen and paper

We’re abstracting a von Neumann machine!
Recap: The “abstraction”
Recap: Popular ISAs

x86

arm

MIPS

RISC-V

Intel

AMD Ryzen

NVIDIA Tegra X1

Apple A13 Bionic

Qualcomm Snapdragon

Broadcom

Swervy Core

Sony Computer Entertainment Inc.
By how much you know about ISA?

• How many of the following is generally true about ISAs?

  ① Many models of processors can support one ISA
  ② An ISA is unique to one model of processor
  ③ Every processor can support multiple ISAs
  ④ Each processor manufacturer has its own ISA

A. 0
B. 1
C. 2
D. 3
E. 4
The abstracted “MIPS” machine

Registers

- $zero
- $at
- $v0
- $v1
- $a0
- $a1
- $a2
- $a3
- $t0
- $t1
- $t2
- $t3
- $t4
- $t5
- $t6
- $t7
- $s0
- $s1
- $s2
- $s3
- $s4
- $s5
- $s6
- $s7
- $t8
- $t9
- $k0
- $k1
- $gp
- $sp
- $fp
- $ra

32-bit

Program Counter

0x0000000000000004

ALU

- add
- addi
- and
- andi
- ori
- xori
- beq
- blt
- jal
- jr

Memory

0x00000000
0x00000008
0x00000010
0x00000018
0x00000020
0x00000028
0x00000030
0x00000038
0xFFFFFFFFC0
0xFFFFFFFFC8
0xFFFFFFFFD0
0xFFFFFFFFD8
0xFFFFFFFFE0
0xFFFFFFFFE8
0xFFFFFFFFF0
0xFFFFFFFFF8

Byte Addressing — every byte of data/instruction has its own address
**MIPS ISA**

- All instructions are **32 bits**
  - 32 32-bit registers
    - All registers are the same
    - $\text{zero}$ is always 0
  - **50** opcodes
    - Arithmetic/Logic operations
    - Load/store operations
    - Branch/jump operations
- **3** instruction formats
  - R-type: all operands are registers
  - I-type: one of the operands is an immediate value
  - J-type: non-conditional, non-relative branches

<table>
<thead>
<tr>
<th>Reg. Name</th>
<th>Reg. Num</th>
<th>Usage</th>
<th>Saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{zero}$</td>
<td>0</td>
<td>zero</td>
<td>N/A</td>
</tr>
<tr>
<td>$\text{at}$</td>
<td>1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$\text{v0-}\text{v1}$</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$\text{a0-}\text{a3}$</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$\text{t0-}\text{t7}$</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$\text{s0-}\text{s7}$</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$\text{t8-}\text{t9}$</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$\text{k0-}\text{k1}$</td>
<td>26-27</td>
<td>OS kernel</td>
<td>no</td>
</tr>
<tr>
<td>$\text{gp}$</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$\text{sp}$</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$\text{fp}$</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$\text{ra}$</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Outline

• Supporting function calls in MIPS
• Other ISA
• Performance Basics
### Subset of MIPS Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1, $s2, 20</td>
<td>$s1 = $s2 + 20</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>and</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>andi</td>
<td>andi $s1, $s2, 20</td>
<td>$s1 = $s2 &amp; 20</td>
</tr>
<tr>
<td></td>
<td>sll</td>
<td>sll $s1, $s2, 10</td>
<td>$s1 = $s2 * 2^10</td>
</tr>
<tr>
<td></td>
<td>srl</td>
<td>srl $s1, $s2, 10</td>
<td>$s1 = $s2 / 2^10</td>
</tr>
<tr>
<td><strong>Data Transfer</strong></td>
<td>lw</td>
<td>lw $s1, 4($s2)</td>
<td>$s1 = mem[$s2+4]</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 4($s2)</td>
<td>mem[$s2+4] = $s1</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if($s1 == $s2), PC = PC + 100</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if($s1 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>

The only type of instructions can access memory.

We use them to support function calls!
Function calls

```c
int main(int argc, char **argv)
{
    n = atoi(argv[0]);
    bar = rand();
    printf("%d\n", foo(n));
    return 0;
}

int foo(n)
{
    int i, sum=0;
    for(i = 0; i < n; i++) {
        sum+=i;
    }
    return sum;
}
```

- **arguments/parameters**: `int argc, char **argv`, `n`, `bar`, `foo(n)`, `int i`, `sum`, `foo(n)`
- **local variables**: `n`, `bar`, `sum`
- **return value**: `0`, `sum`
How these two help to achieve function calls

<table>
<thead>
<tr>
<th>Jump</th>
<th>jal</th>
<th>jal 25</th>
<th>$ra = PC + 4, PC = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>

- Passing arguments
  - $a0-$a3
  - more to go using the **memory stack**
- Invoking the function
  - jal <label>
    - label is the location of the target function
  - store the PC of jal +4 in $ra — why +4?
- Return value in $v0
- Return to caller
  - jr $ra
Sum of natural numbers

```c
#include <stdio.h>

int sum(int n);

int main() {
    int number, result;

    printf("Enter an integer > 0: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum = %d", result);
    return 0;
}

int sum(int n) {
    if (n == 1)
        return 1;
    else
        return n + sum(n-1);
}
```

```assembly
sum:     addi $a0, $a0, -1
         // n = n-1
         jal sum
         // else call sum(n-1)
         add $v0, $v0, $a0
         // ret = sum(n-1)+n-1
         addi $v0, $v0, 1
         // ret += 1
return:  jr $ra
         // return to caller
```

Why doesn’t the program work?

• For the current implementation, the program won’t output the expected solution because?
  ① The program cannot reach the callee, sum
  ② The program cannot pass the arguments correctly
  ③ The program cannot maintain local variables correctly during the expected scope
  ④ The return value of function call will be incorrect
  ⑤ The program cannot return to its caller

A. 0
B. 1
C. 2
D. 3
E. 4
Why doesn’t the program work?

- For the current implementation, the program won’t output the expected solution because?
  ① The program cannot reach the callee, sum
  ② The program cannot pass the arguments correctly
  ③ The program cannot maintain local variables correctly during the expected scope
  ④ The return value of function call will be incorrect
  ⑤ The program cannot return to its caller

A. 0
B. 1
C. 2
D. 3
E. 4

```
main:       ......  
            ......  
            ......  
            add  $a0, $v0, $zero
            jal  sum
PC1:        add  $a1, $zero, $v0
            li   $a0, 4
            syscall
            ......  
            ......  
            ......  
            sum:      addi $a0, $a0, -1
                          addi $v0, $zero, 1
                          beq  $a0, $zero, return
                          jal  sum
                          add  $v0, $v0, $a0
                          addi $v0, $v0, 1
            return:    jr   $ra
```
What happens when we execute the code

Say we input 2!

Is that correct?
Where are we going?
Is $ra pointing to the right place?
Why doesn’t the program work?

- For the current implementation, the program won’t output the expected solution because?
  ① The program cannot reach the callee, sum
  ② The program cannot pass the arguments correctly
  ③ The program cannot maintain local variables correctly during the expected scope
  ④ The return value of function call will be incorrect
  ⑤ The program cannot return to its caller

A. 0
B. 1
C. 2
D. 3
E. 4

```assembly
main:       ...... 
           ...... 
           ...... 
           ...... 
           add $a0, $v0, $zero
           jal  sum
PC1:        add $a1, $zero, $v0
           li   $a0, 4
           syscall
           ...... 
           ...... 
           ...... 
sum:        addi $a0, $a0, -1
           addi $v0, $zero, 1
           beq  $a0, $zero, return
           jal  sum
           add $v0, $v0, $a0
           addi $v0, $v0, 1
return:     jr   $ra
```
What happens when we execute the code

Say we input 2!

Registers

Memory

main: ……
……
……
……
add $a0, $v0, $zero
jal sum
……
li $a0, 4
syscall
……
……
addi $sp, $sp, -8
sw $a0, 0($sp)
sw $ra, 4($sp)

sum: addi $a0, $a0, -1
addi $v0, $zero, 1
beq $a0, $zero, return
jal sum
add $v0, $v0, $a0
addi $v0, $v0, 1
jr $ra

return: lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra

addi $sp, $sp, -8
sw $a0, 0($sp)
sw $ra, 4($sp)

addi $a0, $a0, -1
sw $a0, 0($sp)
sw $v0, 0($sp)
sw $ra, 4($sp)

addi $v0, $zero, 1
jal sum
add $v0, $v0, $a0
addi $v0, $v0, 1
jr $ra

lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra

Say we input 2!
Demo

- The overhead of function calls
- The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead
- Does not work if it’s called using a function pointer
Other ISAs
The abstracted x86 machine

Registers:
- RAX
- RBX
- RCX
- RDX
- RSP
- RBP
- RSI
- RDI
- R8
- R9
- R10
- R11
- R12
- R13
- R14
- R15
- RIP
- FLAGS
- CS
- SS
- DS
- ES
- FS
- GS

Memory:
- 0x0000000000000000
- 0x0000000000000008
- 0x0000000000000010
- 0x0000000000000018
- 0x0000000000000020
- 0x0000000000000028
- 0x0000000000000030
- 0x0000000000000038

ALU:
- ADD
- SUB
- IMUL
- AND
- OR
- XOR

Jump:
- JMP
- JE
- CALL
- RET

MOV

64-bit

2^{64} Bytes
x86 ISA

• The most widely used ISA

• A poorly-designed ISA
  • It breaks almost every rule of a good ISA
    • variable length of instructions
    • the work of each instruction is not equal
    • makes the hardware become very complex
  • It’s popular != It’s good

• You don’t have to know how to write it, but you need to be able to read them and compare x86 with other ISAs

• Reference
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
# Registers

<table>
<thead>
<tr>
<th>16bit</th>
<th>32bit</th>
<th>64bit</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
<td>The accumulator register</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
<td>The base register</td>
<td>These can be used more or less interchangeably</td>
</tr>
<tr>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
<td>The counter</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
<td>The data register</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
<td>Pointer to the base of stack frame</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>RnD</td>
<td></td>
<td>General purpose registers (8-15)</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
<td>Source index for string operations</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
<td>Destination index for string operations</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>EIP</td>
<td>RIP</td>
<td>Instruction pointer</td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td></td>
<td></td>
<td>Condition codes</td>
<td></td>
</tr>
</tbody>
</table>
MOV and addressing modes

- MOV instruction can perform load/store as in MIPS
- MOV instruction has many address modes
  - an example of non-uniformity

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $6, %eax</td>
<td>R[eax] = 0x6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl .L0, %eax</td>
<td>R[eax] = .L0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl %ebx, %eax</td>
<td>R[ebx] = R[eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl -4(%ebp), %ebx</td>
<td>R[ebx] = mem[R[ebp]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl (%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>movl -4(%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4-4]</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>movl %ebx, -4(%ebp)</td>
<td>mem[R[ebp]-4] = R[ebx]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl $6, -4(%ebp)</td>
<td>mem[R[ebp]-4] = 0x6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Addressing and accessing the data structure

• Memory allocation
  • Each object-instance of the data structure occupies consecutive memory locations that can accommodate all members in this object-instance
  • The starting address of each object-instance must be aligned with the multiple of 8 — 64-bit
    • Try to have as many members aligned with address multiplied by 8 using the smallest amount of space, but also maintains the member order
    • Although ARM supports unaligned access — they are slow

• Memory access:
  • The base address register points to the beginning of the accessing object-instance
  • The offset points to the member — one of the reason why we have an offset field
The result of `sizeof(struct student)`

- Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What's the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20
B. 28
C. 32
D. 36
E. 40
The result of `sizeof(struct student)`

- Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What’s the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20
B. 28
C. 32
D. 36
E. 40
Memory layout of data structures

struct node {
    int data;
    struct node *next;
};
The result of `sizeof(struct student)`

- Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What’s the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20
B. 28
C. 32
D. 36
E. 40
Arithmetic Instructions

- Accepts memory addresses as operands
- Register-memory ISA

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subl $16, %esp</code></td>
<td>$R[%esp] = R[%esp] - 16$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>subl %eax, %esp</code></td>
<td>$R[%esp] = R[%esp] - R[%eax]$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>subl -4(%ebx), %eax</code></td>
<td>$R[\text{eax}] = R[\text{eax}] - \text{mem}[R[\text{ebx}]-4]$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><code>subl (%ebx, %edx, 4), %eax</code></td>
<td>$R[\text{eax}] = R[\text{eax}] - \text{mem}[R[\text{ebx}]+R[\text{edx}]*4]$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><code>subl -4(%ebx, %edx, 4), %eax</code></td>
<td>$R[\text{eax}] = R[\text{eax}] - \text{mem}[R[\text{ebx}]+R[\text{edx}]*4-4]$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><code>subl %eax, -4(%ebx)</code></td>
<td>$\text{mem}[R[\text{ebx}]-4] = \text{mem}[R[\text{ebx}]-4]-R[\text{eax}]$</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Branch instructions

- x86 use condition codes for branches
  - Arithmetic instruction sets the flags
  - Example:
    \[
    \text{cmp } \%eax, \%ebx \# \text{computes } \%eax-\%ebx, \text{ sets the flag}
    \]
    \[
    \text{je } <\text{location}> \# \text{jump to location if equal flag is set}
    \]

- Unconditional branches
  - Example:
    \[
    \text{jmp } <\text{location}> \# \text{jump to location}
    \]
Translate from C to Assembly

- gcc: gcc [options] [src_file]
  - compile to binary
    - gcc -o foo foo.c
  - compile to assembly (assembly in foo.s)
    - gcc -S foo.c
  - compile with debugging message
    - gcc -g -S foo.c
  - optimization
    - gcc -On -S foo.c
      - n from 0 to 3 (0 is no optimization)
Demo

- The magic of compiler optimization!
- Without optimization
- After compiled with `-O3`
Summation for x86

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
xorl %eax, %eax
.L2: addl (%ecx,%eax,4), %edx
     addl $1, %eax
     cmpl $100, %eax
     jne .L2
```

Assume
int is 32 bytes
%ecx = &A[0]
%edx = sum;
%eax = i;
```
Comparing x86 and MIPS ISAs, how many of the following statements is/are “generally” correct?

① x86 provides more instructions than MIPS
② x86 usually needs more instructions to express the same program
③ An x86 instruction may access memory for 3 times
④ An x86 instruction may be shorter than a MIPS instruction
⑤ An x86 instruction may be longer than a MIPS instruction

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
Comparing x86 and MIPS ISAs, how many of the following statements is/are “generally” correct?

① x86 provides more instructions than MIPS
② x86 usually needs more instructions to express the same program
③ An x86 instruction may access memory for 3 times
④ An x86 instruction may be shorter than a MIPS instruction
⑤ An x86 instruction may be longer than a MIPS instruction

A. 1
B. 2
C. 3
D. 4
E. 5
Comparing x86 and MIPS ISAs, how many of the following statements is/are “generally” correct?

① x86 provides more instructions than MIPS
② x86 usually needs more instructions to express the same program
③ An x86 instruction may access memory for 3 times
④ An x86 instruction may be shorter than a MIPS instruction
⑤ An x86 instruction may be longer than a MIPS instruction

A. 1
B. 2
C. 3
D. 4
E. 5
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td>base+offset base+index scaled+index scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Popular ISAs

Complex Instruction Set Computers (CISC)

Reduced Instruction Set Set Computers (RISC)

x86

arm

MIPS

RISC-V
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td>base+offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
<tr>
<td>ISA type</td>
<td>Reduced Instruction Set Computers (RISC)</td>
<td>Complex Instruction Set Computers (CISC)</td>
</tr>
</tbody>
</table>
How many operations: CISC v.s. RISC

• CISC (Complex Instruction Set Computing)
  • Examples: x86, Motorola 68K
  • Provide many powerful/complex instructions
    • Many: more than 1503 instructions since 2016
    • Powerful/complex: an instruction can perform both ALU and memory operations

• RISC (Reduced Instruction Set Computer)
  • Examples: ARMv8, RISC-V, MIPS (the first RISC instruction, invented by the authors of our textbook)
  • Each instruction only performs simple tasks
  • Easy to decode
Announcements

• Assignment #1 is up!
  • Due next Monday before the lecture
  • Check the website for the template
  • Submit a pdf through Canvas
  • TA’s discussion this Thursday will give you hints on these questions
• Reading quizzes
  • Due next Tuesday
• Resources
  • Ask questions — piazza
  • Reading quizzes, turning in assignments — Canvas
  • Slides, schedule, assignment questions — Check our website
  • Video archive — Prof. Usagi’s Youtube channel