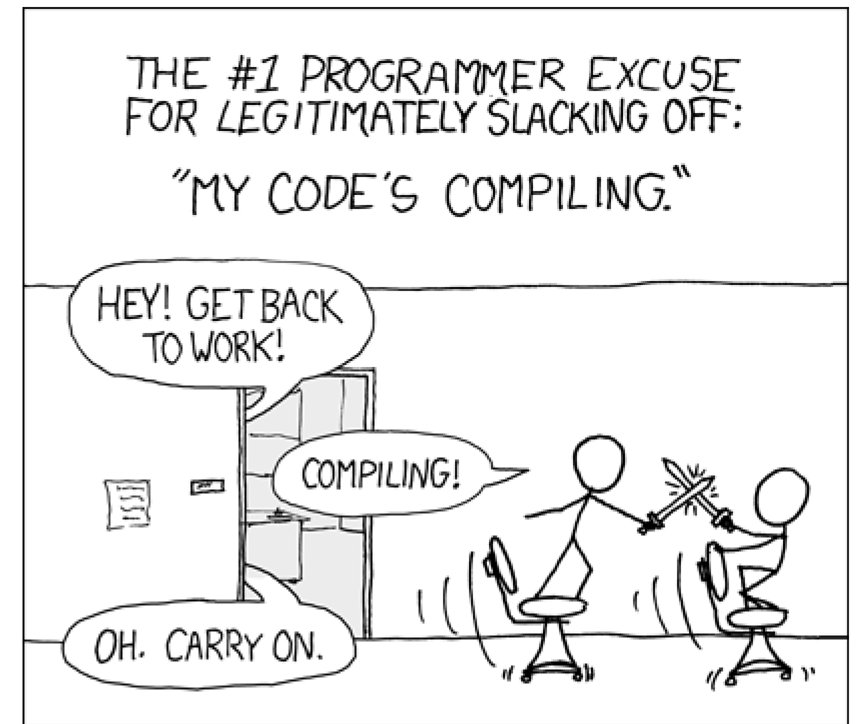


https://imgs.xkcd.com/comics/compiling.png

CMPT 295

Unit - Instruction Set Architecture

Lecture 25 – ISA Design + Evaluation

Last Lecture

- Looked at an example of a RISC instruction set: MIPS
  - From its **Instruction Set Architecture (ISA):**
    - Registers
    - Memory model
    - (Sub)set of instructions
      - Assembly instructions
      - Machine instructions
        - Format of *R-format* of a MIPS machine instruction
          - Size of its fields

Format of assembly instruction not necessarily == format of machine instruction

From last lecture!

Example of an ISA: MIPS

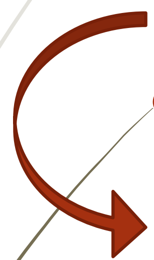
- Function call conventions
  - **caller** saved registers
  - **callee** saved registers
- Model of Computation
  - Sequential

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

From last lecture!

MIPS - Design guidelines

3. In terms of machine instruction format:
   a. Create as few of them as possible
   b. Have them all of the same length and same format!
   c. If we have different machine instruction formats, then position the **fields** that have the **same purpose** in the **same location** in the format



- **Can all MIPS machine instructions have the same length and same format?**
  - For example: `lw $s1, 20($s2)` =>

opcode	rs	rt	rd	shamt	func

?

  - When designing its corresponding machine instruction …
    - Must specify source register using 5 bits -> OK!
    - Must specify destination register using 5 bits -> OK!
    - Must specify a constant using 5 bits -> Hum…
      - Value of constant limited to $[0..2^5-1]$
      - Often use to access array elements so needs to be > $2^5 = 32$

From last lecture!

MIPS ISA designers compromise

- Keep all machine instructions format the same length
- Consequence -> different formats for different kinds of MIPS instructions
  - *R-format* for register
  - *I-format* for immediate
  - *J-format* for jump

opcode 6 bits	rs 5 bits	rt 5 bits	rd 5 bits	shamt 5 bits	func 6 bits
opcode 6 bits	rs 5 bits	rt 5 bits	Address/immediate 16 bits		
opcode 6 bits	Target address 26 bits				

- **opcode** indicates the instruction as well as the format of the instruction
  - This way, the hardware knows whether to treat the last half of the instruction as 3 fields (*R-format*) or as 1 field (*I-format*)
- Since we have different machine instruction formats, **fields** with **same purpose** are positioned in the **same location** in the 3 formats ☺

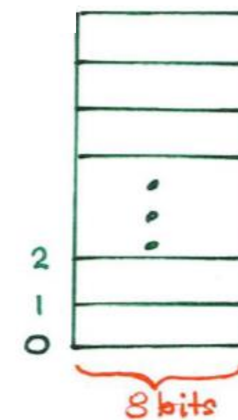
from c. on previous slide

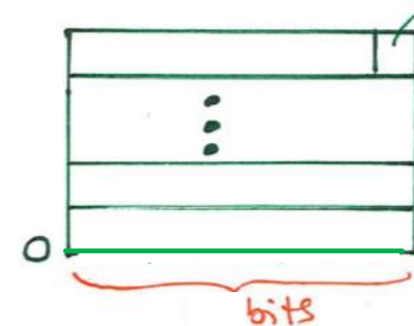
Today's Menu

- Instruction Set Architecture (ISA)
  - Definition of ISA
- Instruction Set design
  - Design guidelines
  - Example of an instruction set: MIPS
  - Create our own instruction sets
  - ISA evaluation
- Implementation of a microprocessor (CPU) based on an ISA
  - Execution of machine instructions (datapath)
  - Intro to logic design + Combinational logic + Sequential logic circuit
  - Sequential execution of machine instructions
  - Pipelined execution of machine instructions + Hazards

Let's design our own ISA - **x295M** (1 of 2)

- Registers and Memory model
  - # of registers -> 0 registers – model called "Memory Only" (except **$sp**)
  - Each memory address has -> 32 bits (m = 32)
  - Word size -> 32 bits
  - Byte-addressable memory
    so address resolution -> n = 1 byte (8 bits)
  - Memory size -> 2^m x n -> 2^{32} × 1 byte
    OR 2^{32} × 8 bits



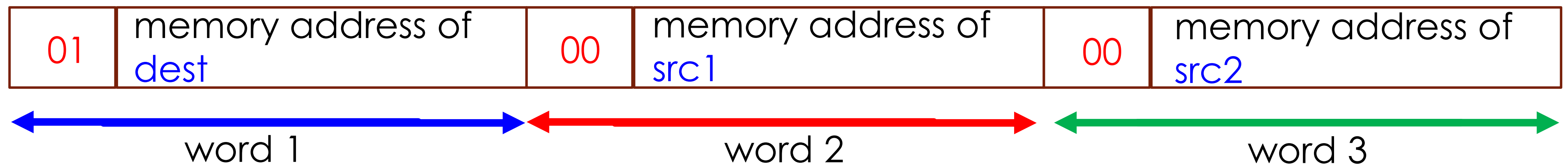


Let's design our own ISA - **x295M** (2 of 2)

- (Sub)set of instructions

x295M assembly language instructions	Semantic (i.e., Meaning)	Corresponding x295M machine instructions
`ADD a, b, c`	`M[c] = M[a] + M[b]`	01 <30 bits> 00 <30 bits> 00 <30 bits>
`SUB a, b, c`	`M[c] = M[a] - M[b]`	10 <30 bits> 00 <30 bits> 00 <30 bits>
`MUL a, b, c`	`M[c] = M[a] * M[b]`	11 <30 bits> 00 <30 bits> 00 <30 bits>

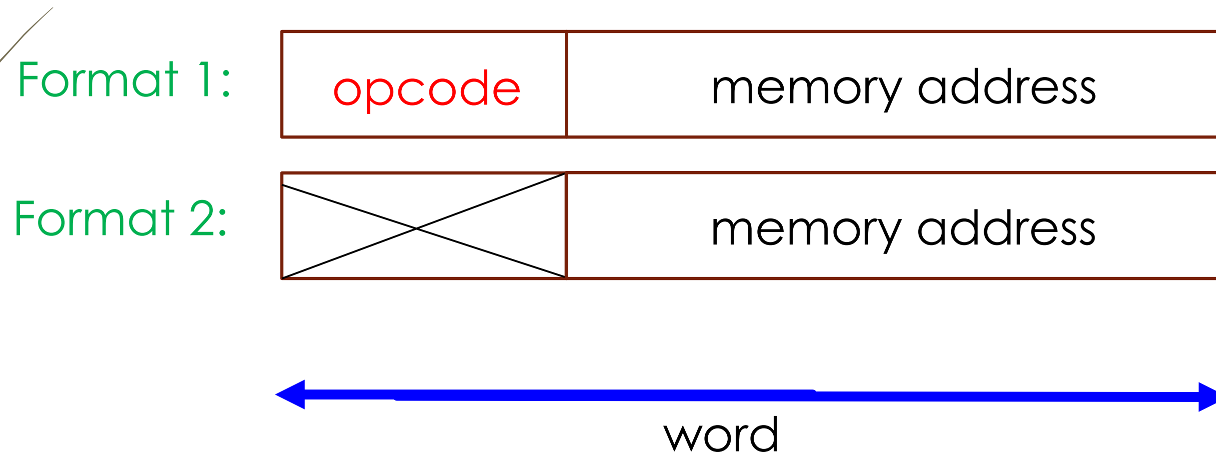
- Memory addressing mode -> **Direct (absolute), base + displacement and indirect**
- Operand model -> "3 Operand" model
- Format of corresponding **x295M** machine instructions:



- Size of opcode -> 2 bits   Size of operand -> 30 bits
- Length of 1 instruction -> __________ bits

Note about the machine code format

- This ISA has two machine code formats:



About Design guideline

3. In terms of machine instruction format:

a. Create as few of them as possible -> 2 formats

b. Have them all of the same length -> 32 bits

c. Since we have two different machine instruction formats, **fields** with **same purpose** are positioned in the **same location** in the 2 formats

-> operand field (purpose -> memory address) positioned in the same location in the 2 formats

Evaluation of our ISA **x295M** versus **MIPS**

- Sample C code: `z = (x + y) * (x - y)`

C code -> Assembly code

```
ADD 0($sp), 4($sp), 12($sp)
SUB 0($sp), 4($sp), 16($sp)
MUL 12($sp), 16($sp), 8($sp)
```

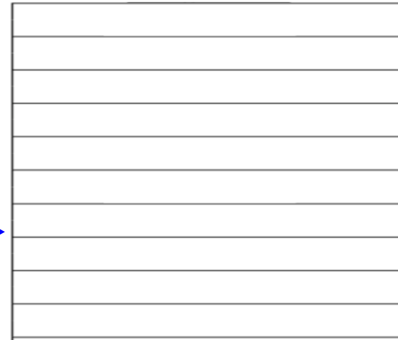
Meaning

```
M[$sp+12] = M[$sp+0] + M[$sp+4]
M[$sp+16] = M[$sp+0] – M[$sp+4]
M[$sp+8] = M[$sp+12] * M[$sp+16]
```

Evaluation of our ISA **x295M** versus **MIPS**

- Sample C code: `z = (x + y) * (x - y)`

Stack Segment



0x7fffff00<- $sp->

Memory

Text Segment

0x00400000 ->

Opcode	Encoding
ADD	01
SUB	10
MUL	11
No op	00

Assembly code | Memory address of each instruction | **Machine code**

Assembly code	0x00400000	0x00400004	0x00400008
ADD 0($sp), 4($sp), 12($sp)->	01 0x7fffff0c	00 0x7fffff00	00 0x7fffff04

Assembly code	0x0040000c	0x00400010	0x00400014
SUB 0($sp), 4($sp), 16($sp)->	10 0x7fffff10	00 0x7fffff00	00 0x7fffff04

Assembly code	0x00400018	0x0040001c	0x00400020
MUL 12($sp), 16($sp), 8($sp)->	11 0x7fffff08	00 0x7fffff0c	00 0x7fffff10

Evaluation of our ISA **x295M** versus **MIPS**

- Sample C code: `z = (x + y) * (x - y)`

C code -> Assembly code	Meaning
`lw $s1, 0($sp)`	`$s1 = M[$sp + 0]`
`lw $s2, 4($sp)`	`$s2 = M[$sp + 4]`
`add $s3, $s1, $s2`	`$s3 = $s1 + $s2`
`sub $s4, $s1, $s2`	`$s4 = $s1 - $s2`
`mul $s5, $s3, $s4`	`$s5 = $s3 * $s4`
`sw $s5, 8($sp)`	`M[$sp + 8] = $s5`

Evaluation of our ISA **x295M** versus **MIPS**

Opcode + func	Encoding
lw	35_{10}
sw	43_{10}
add	$0 + 32_{10}$
sub	$0 + 34_{10}$
mul	$0 + 36_{10}$

Register	Number
$s1	17_{10}
$s2	18_{10}
$s3	19_{10}
$s4	20_{10}
$s5	21_{10}
$sp	29_{10}

- Sample C code: `z = (x + y) * (x - y)`

I-format **src** **dest**

opcode 6 bits	rs 5 bits	rt 5 bits	Address/immediate 16 bits

Assembly code	**Machine code**
`lw $s1, 0($sp)->`	`100011 11101 10001 0x0000`
`lw $s2, 4($sp)->`	`100011 11101 10010 0x0004`
`sw $s5, 8($sp)->`	`101011 10101 11101 0x0008`

R-format **src1** **src2** **dest**

opcode 6 bits	rs 5 bits	rt 5 bits	rd 5 bits	shamt 5 bits	func 6 bits

Assembly code	**Machine code**
`add $s3, $s1, $s2->`	`000000 10001 10010 10011 00000 100000`
`sub $s4, $s1, $s2->`	`000000 10001 10010 10100 00000 100010`
`mul $s5, $s3, $s4->`	`000000 10011 10100 10101 00000 100100`

Evaluation of our ISA x295M versus MIPS

- Sample C code: `z = (x + y) * (x - y)`

Memory address of each instruction

Address	x295M in machine code
0x00400000	01 11 1111 1111 1111 1111 1111 0000 1100
0x00400004	00 11 1111 1111 1111 1111 1111 0000 0000
0x00400008	00 11 1111 1111 1111 1111 1111 0000 0100
0x0040000c	10 11 1111 1111 1111 1111 1111 0001 0000
0x00400010	00 11 1111 1111 1111 1111 1111 0000 0000
0x00400014	00 11 1111 1111 1111 1111 1111 0000 0100
0x00400018	11 11 1111 1111 1111 1111 1111 0000 1000
0x0040001c	00 11 1111 1111 1111 1111 1111 0000 1100
0x00400020	00 11 1111 1111 1111 1111 1111 0001 0000

x295M in machine code

Memory address of each instruction

Address	MIPS in machine code
0x00400000	100011 11101 10001 0000 0000 0000 0000
0x00400004	100011 11101 10010 0000 0000 0000 0100
0x00400008	000000 10001 10010 10011 00000 100000
0x0040000c	000000 10001 10010 10100 00000 100010
0x00400010	000000 10011 10100 10101 00000 100100
0x00400014	101011 10101 11101 0000 0000 0000 0008

MIPS in machine code

Which criteria shall we use when comparing/evaluating ISAs?

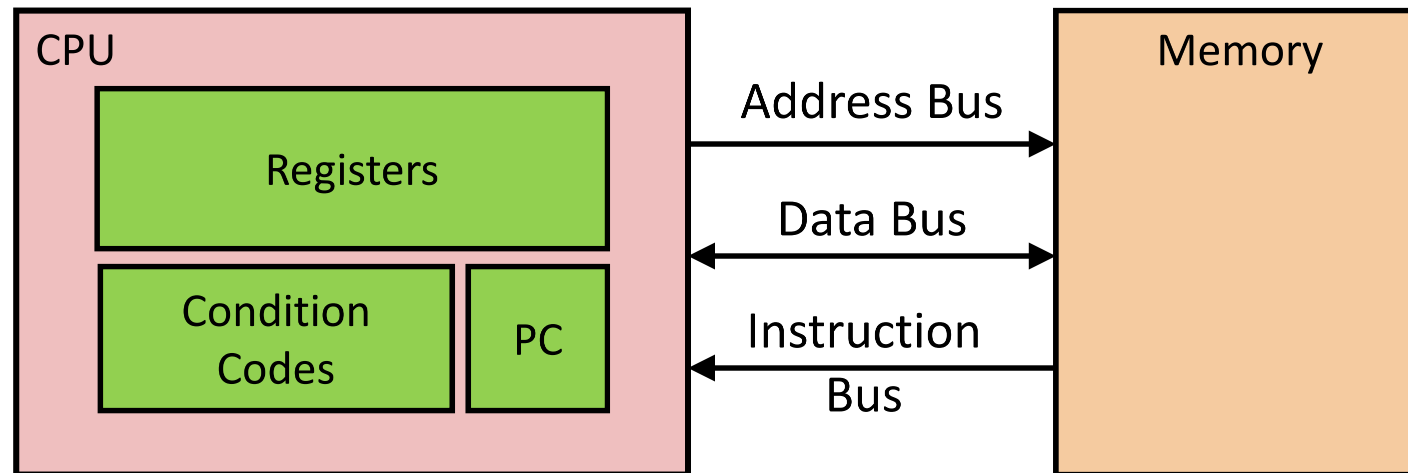
- Whether or not the Instruction set (**IS**) design guidelines have been satisfied:
  1. Each instruction of **IS** have an unambiguous **binary encoding**
  2. **IS** is functionally complete -> i.e., it is “Turing complete”
  3. In terms of machine instruction format:
     a. Create as few of them as possible
     b. Have them all of the same length
     c. If we have different machine instruction formats, then position the **fields** that have the **same purpose** in the **same location** in the format

Which criteria shall we use when comparing/evaluating ISAs?

- Program performance -> usually measured using time
  - If an ISA design results in faster program execution then it is deemed "better"
- What can affect the time a program takes to execute?
  - Since accessing memory is slow (slower than accessing registers), the number of memory accesses a program does will affect its execution time
  - Therefore, possible criteria: **number of memory accesses**
  - The fewer memory accesses our program makes, the faster it executes, hence the "better" it is

Why is memory access slow!

- Memory access is the most time constraining aspect of program execution
- Why? Because of transfer rate limitation of the bus between memory and CPU
  - Memory is "far away" from the CPU so it takes time to transfer instructions and data from memory to microprocessor
- This is known as the **von Neumann bottleneck**



- From the above diagram, we can gather that register access is faster than memory access! Why?

How is the **von Neumann Bottleneck** created?

- It is created when **memory is accessed**
- During **fetch** stage
  - An instruction is retrieved from memory
- During **decode/execute** stages
  - The value of operands may be read from memory
  - The result may be written to memory

Evaluation of our ISA **x295M** versus **MIPS**

- Sample C code: `z = (x + y) * (x - y)`
- Let's count the **number of memory accesses**:

x295M	fetch	decode/ execute
`ADD 0($sp), 4($sp), 12($sp)`		
`SUB 0($sp), 4($sp), 16($sp)`		
`MUL 12($sp), 16($sp), 8($sp)`		
Total:		

MIPS	fetch	decode/ execute
`lw  $s1, 0($sp)`		
`lw  $s2, 4($sp)`		
`add $s3, $s1, $s2`		
`sub $s4, $s1, $s2`		
`mul $s5, $s3, $s4`		
`sw  $s5, 8($sp)`		

Summary

- ISA design
  - **`MIPS`**
  - Created our own **`x295M`**: “Memory only”
- ISA Evaluation
  - Examining the effect of the **von Neumann bottleneck** on the execution time of our program by counting **number of memory accesses**
    - The fewer memory accesses our program makes, the faster it executes, hence the “better” it is
- Improvements:
  - Decreasing effect of **von Neumann bottleneck** by reducing the **number of memory accesses**

Next Lecture

- Instruction Set Architecture (ISA)
  - Definition of ISA
- Instruction Set design
  - Design principles
  - Look at an example of an instruction set: MIPS
  - Create our own
  - ISA evaluation
- Implementation of a microprocessor (CPU) based on an ISA
  - Execution of machine instructions (datapath)
  - Intro to logic design + Combinational logic + Sequential logic circuit
  - Sequential execution of machine instructions
  - Pipelined execution of machine instructions + Hazards