Machine Language

COMP375

Computer Architecture and Organization

Bunch of Bytes

- Machine language is binary codes that the computer executes.
- The computer fetches the instructions from memory and executes them.

Machine language for a square root program

```
8b 45 e0 89 45 f8 89 45 ec 8b 45 ec
89 45 f8 8b 45 e0 ba 00 00 00 00 f7
7d f8 03 45 f8 d1 f8 89 45 ec 3b 45
f8 75 e2 8b f4
```

Assembler and Machine

- Assembler language is the easy way to write machine language.
- Each line of an assembler program generates one machine language instruction.
- The assembler allows you to use variable names instead of numerical addresses and instruction mnemonics instead of numerical operation codes.

Instruction Format

- The general format for a machine language instruction is

The operands can be a memory address, a register or a value.

Op codes

- Each assembler instruction represents a numerical machine language opcode.

add	05
cmp	3B
dec	FF
idiv	7F
jmp	39
push	68
sar	D0

Data Location

- **Register** – The data is in a CPU register.
- **Memory** – The data is in a location in RAM
- **Immediate** – The data is part of the instruction. Immediate data items are read-only.

Intel Assembler

- The Intel assembler allows you to use one mnemonic for different op codes.
- There are several versions of the add instruction based on the size of the operands. The assembler picks the correct op code.

Opcode	Instruction	Description
04 ib	ADD AL,imm8	ADD imm8 to AL
05 iw	ADD AX,imm16	ADD imm16 to AX
05 id	ADD EAX,imm32	ADD imm32 to EAX
80 /0 ib	ADD r/m8,imm8	ADD imm8 to r/m8
81 /0 iw	ADD r/m16,imm16	ADD imm16 to r/m16
81 /0 id	ADD r/m32,imm32	ADD imm32 to r/m32
83 /0 ib	ADD r/m16,imm8	ADD sign-extended
83 /0 ib	ADD r/m32,imm8	ADD sign-extended i
00 /r	ADD r/m8,r8	ADD r8 to r/m8
01 /r	ADD r/m16,r16	ADD r16 to r/m16
01 /r	ADD r/m32,r32	ADD r32 to r/m32
02 /r	ADD r8,r/m8	ADD r/m8 to r8
03 /r	ADD r16,r/m16	ADD r/m16 to r16
03 /r	ADD r32,r/m32	ADD r/m32 to r32

Mnemonic to Op Code Mapping

- Intel assembler uses the same mnemonic for the machine language instruction to:
  - Move a byte from memory to a register
  - Move a byte from a register to memory
- The Intel `mov` instruction generates different machine language op codes depending upon the size of the operands.

Number of Operands

- In addition to the op code, the instruction might contain zero, one, two or three operands.
- Different architectures use different number of operands.
- A single architecture may have instructions with differing number of operands.

No operand instructions

- Some instructions do not require any operands. The data affected is implied in the instruction.
- **`RET`** — Return from function
- **`HLT`** — Halt
- **`CPUID`** — Get details about the CPU
- **`LAHF`** — Load Status Flags into AH Reg

One Operand Instructions

- Some unary operations require only one operand

`inc al` - increment the al register
`jmp address` – jump to the address

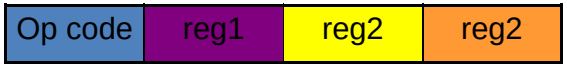
Two Operand Instructions

- Many instructions act on two operands
- Most math instructions use two operands and return the results in one of them.
- **`add al,varname`**
- **`add bl,al`**
- **`imul eax, varname`**

Three Operand Instructions

- Some machines support three operands (Intel Pentium does not).
- Most MIPS instructions use three operands

```
add     R1,R2,R3
```

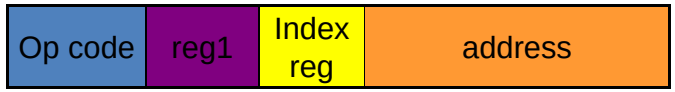


Additional Instruction Fields

- Most architectures support an addressing mode that combines an address field in the instruction and the contents of a register

```
add     R3,addr[R7]
```

- This instruction adds the contents of register R3 with the memory location whose address is the sum of the address field and R7.



Variable or Fixed Length

- Some architectures use variable length instructions. Instructions with more operands or memory addresses are longer.
  - saves memory
- Some architectures always use the same length instruction.
  - easier to find the beginning of instructions
  - instructions are in aligned words

Assembled Code

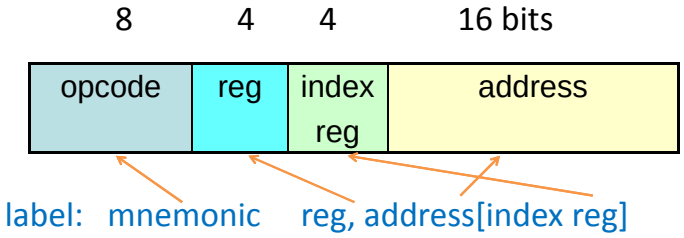
```
addr  machine                  assembler
004a  8b 45 e0           mov   eax, number[ebp]
004d  89 45 f8           mov   good[ebp], eax
0050  89 45 ec           mov   better[ebp], eax
0053  8b 45 ec           mov   eax, better[ebp]
                   again:
0056  89 45 f8           mov   good[ebp], eax
0059  8b 45 e0           mov   eax, number[ebp]
005c  ba 00 00 00 00     mov   edx, 0
0061  f7 7d f8           idiv  good[ebp]
0064  03 45 f8           add   eax, good[ebp]
0067  d1 f8              sar   eax, 1
0069  89 45 ec           mov   better[ebp], eax
006c  3b 45 f8           cmp   eax, good[ebp]
006f  75 e2              jne   SHORT again
0071  8b f4              mov   esi, esp
```

Questions about the code

- What does SHORT mean after the `jne` ?
- What is the displacement of the `jne` ?

Example Machine Language

- Assume each instruction of this imaginary computer is 32 bits in length



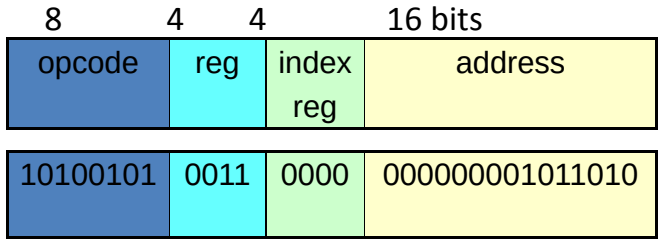
Assembler language format

Machine Language Program

```
00   01100018    LOAD   R1, y
04   2110001C    DIV    R1, z
08   05100020    ADD    R1, five
0C   02100014    STORE  R1, x
10   47000000    RET
14               x      res
18               y      res
1C               z      res
20   00000005    five   +5
```

World of Numbers

- The opcodes are numbers
- The address is a number
- The register field is a number



Add R3, *[no indexing]* xyz

Disassembling

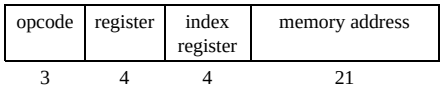
- Any bunch of bits in memory can be considered a program.
- Most random bits may not produce a logical program and may generate errors due to bad opcodes or addressing errors.
- A disassembler is a program that interprets the values in memory as instructions.
- Some software asks not to be disassembled

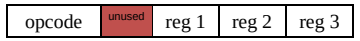
What are the Instructions?

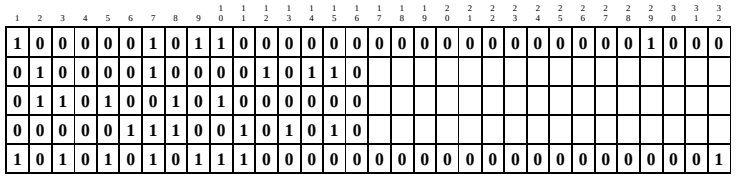
Opcodes

Add	Subtract	Multiply	Divide	Load	Store	JumpEql	Jump
0	1	2	3	4	5	6	7

Instruction Formats





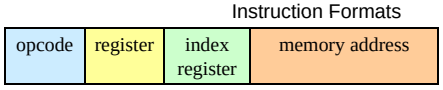


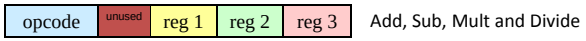
What are the Instructions?

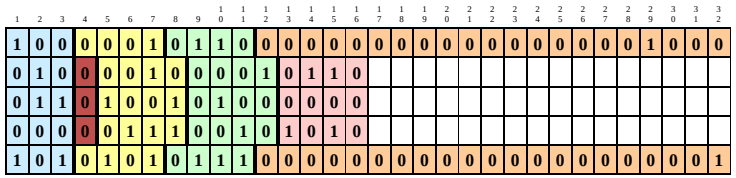
Opcodes

Add	Subtract	Multiply	Divide	Load	Store	JumpEql	Jump
0	1	2	3	4	5	6	7

Instruction Formats







Notes on the Example Architecture

- This is an example of a “Load/Store” architecture. Only the load and store instructions access memory.
- Why is there a one bit unused field in the arithmetic instructions?
- Why is the opcode always the left most bits?
- The format of the jump instructions was not shown. What might be a good format?

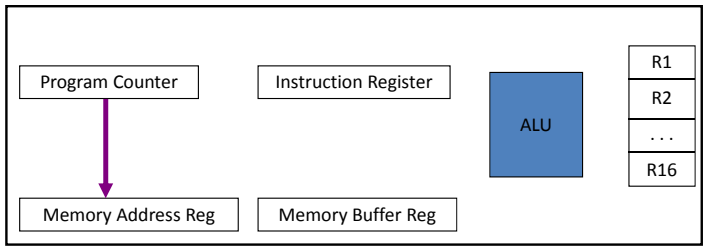
Instruction Cycle

- Fetch the instruction from the memory address in the Program Counter register
- Increment the Program Counter
- Decode the type of instruction
- Fetch the operands
- Execute the instruction
- Store the results

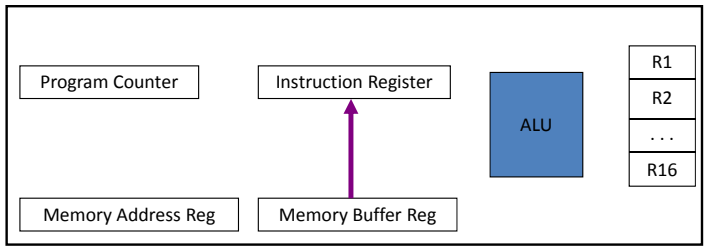
Basic Processor Components

- **Program Counter** – contains the address of the next instruction to execute.
- **Arithmetic Logic Unit** – logic to perform arithmetic and logical functions
- **User registers** – hold data
- **Memory Address Register** – contains the address to be copied to or from RAM
- **Memory Buffer Register** – contains data copied to or from RAM.

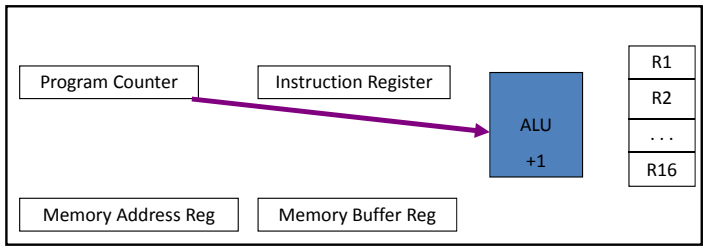
Instruction Fetch



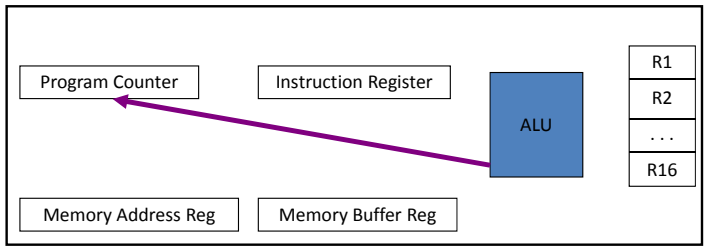
Instruction Fetch



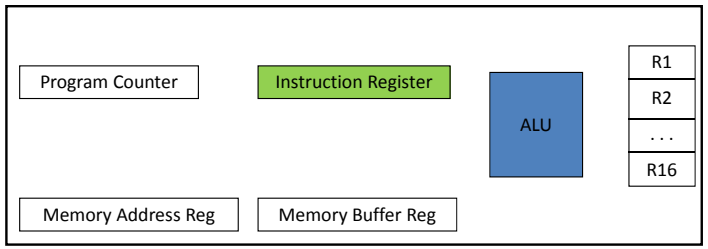
Increment Program Counter



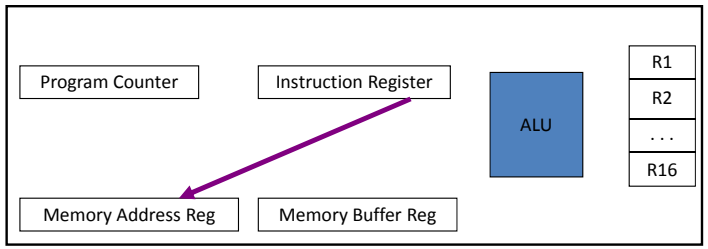
Increment Program Counter



Decode Instruction

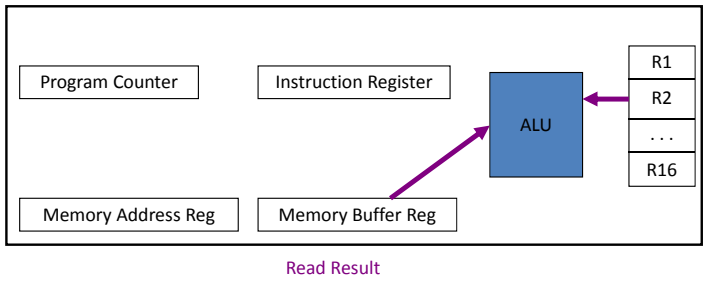


Operand Fetch

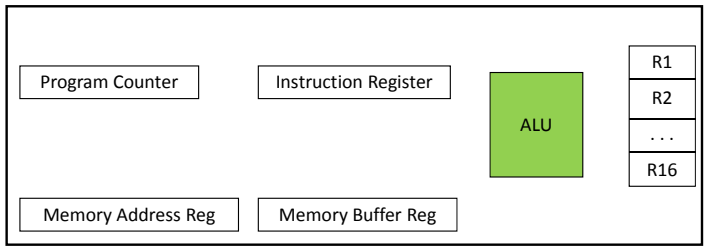


Memory Read

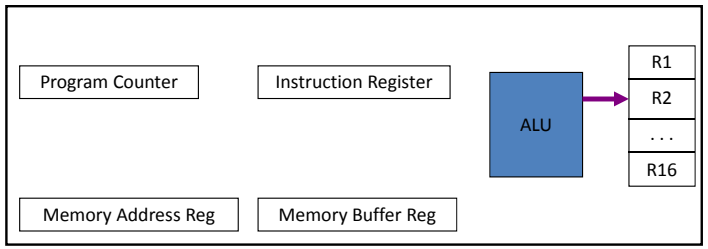
Operand Fetch



Execution



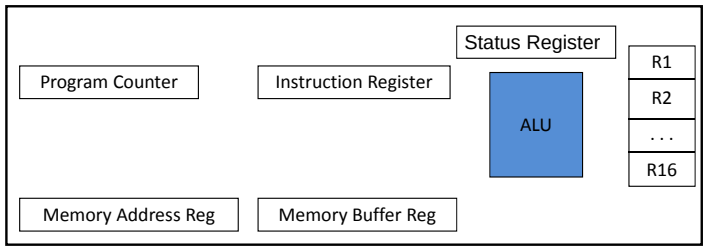
Result Store



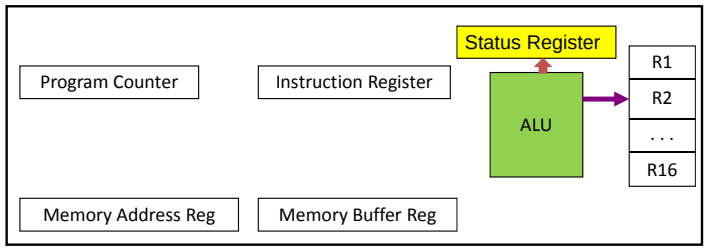
Jump Instruction

- Consider an arithmetic instruction followed by a jump instruction.
- The arithmetic instruction sets bits in the status register

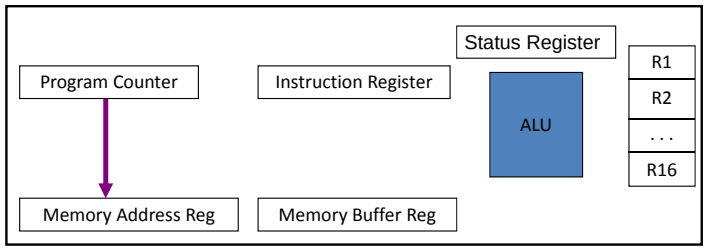
Execution Stage of Instruction 1



Result Save Stage of Instruction 1

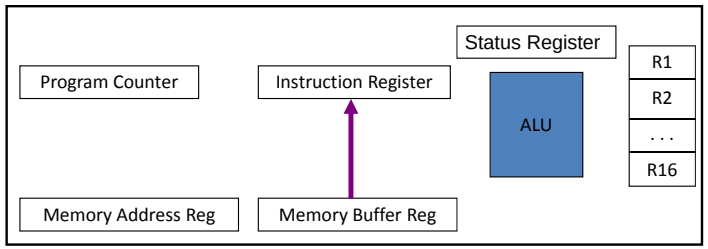


Instruction 2 Fetch



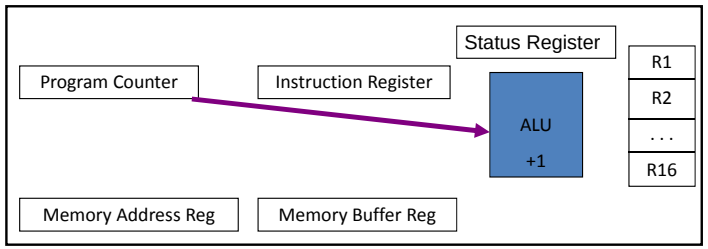
Memory Read

Instruction 2 Fetch

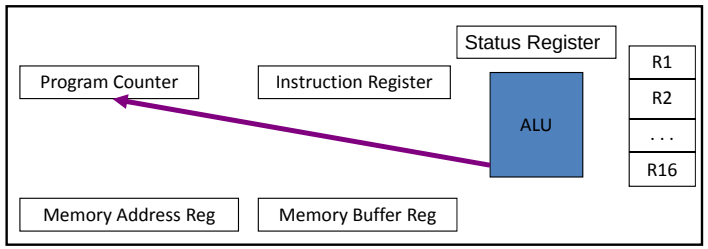


Read Result

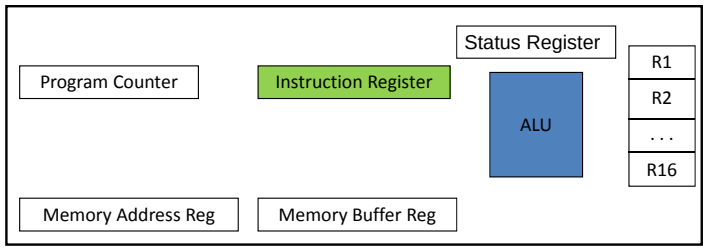
Increment Program Counter



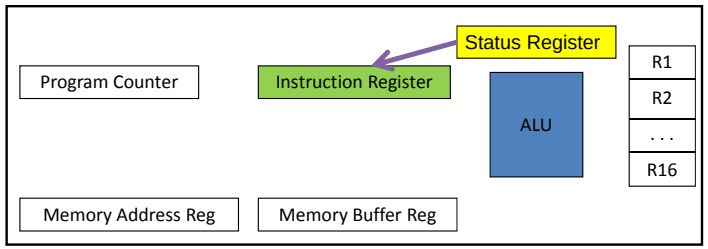
Increment Program Counter



Decode Instruction



Execution of Instruction 2



Saving Result of Instruction 2