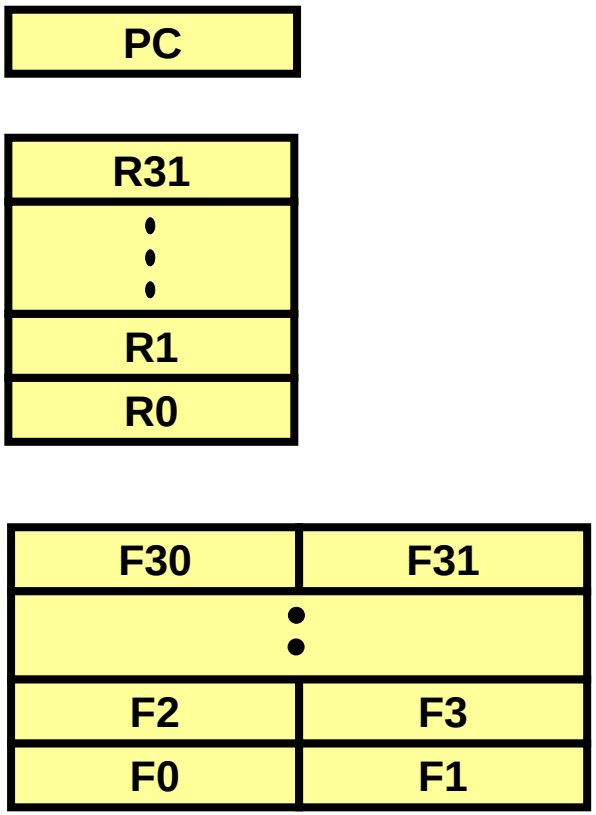
Lecture 6: Instruction Set Architectures - III

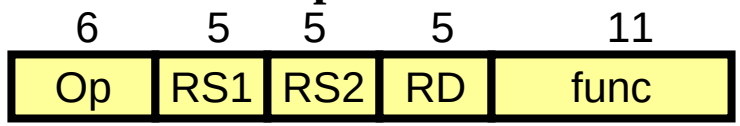
- Last Time
  - Instruction types
  - Addressing modes
  - Control flow operations
  - Instruction formats
  - Case study ISAs – MIPS, graphics processors

- Today
  - Register and Memory
  - Exceptions
  - ISA principles
  - Interaction between ISA and compiler

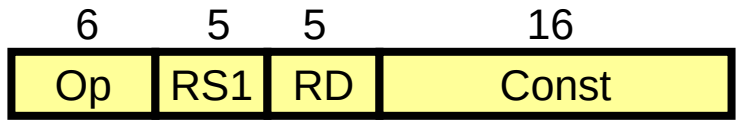
MIPS ISA (a visual)



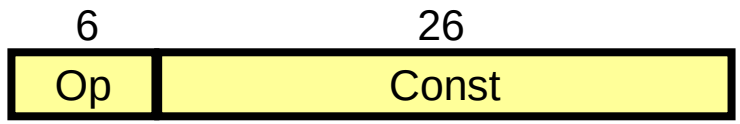
R: rd ← rs1 op rs2



I: ld/st, rd ← rs1 op imm, branch



J: j, jal



Fixed-Format

MIPS Instruction Types

- **ALU Operations**
  - arithmetic – int and float (`add, sub, mult`)
  - logical (`and, or, xor, srl, sra`)
  - data type conversions (`cvt.w.d, cvt.s.d`)
- **Data Movement**
  - memory reference (`lb, lw, sb, sw`)
  - register to register (`move, mfhi`)
- **Control** - what instruction to do next
  - tests/compare (`slt, seq`)
  - branches and jumps (`beq, bne, j, jr`)
  - support for procedure call (`jal, jalr`)
  - operating system entry (`syscall`)

Naming storage locations

Naming Storage in ISAs

- Memory
  - Addresses in instruction
  - Addresses computed by instructions
- General Registers
  - Operands to instructions
- Special registers
  - Status, condition codes, floating-point codes
  - Operands to special instructions

How many names (operands) per instruction?

• No Operands	HALT NOP	
• 1 operand	NOT R4 JMP _L1	R4 ⇐ R4
• 2 operands	ADD R1, R2 LDI R3, #1234	R1 ⇐ R1 + R2
• 3 operands	ADD R1, R2, R3	R1 ⇐ R2 + R3
• > 3 operands	MADD R4,R1,R2,R3	R4 ⇐R1+(R2*R3)

Two ways to specify an operand

1) We don't - operands can be implicit

   Example: 'RET' on x86 architecture
   (return address implicitly at top of stack)

2) Actual value – e.g. 0x3f as an immediate value in instruction

3) Indirectly, using an operand specifier.

   Two parts to an operand specifier:

   b) Namespace (usually implicit) – e.g 'registers'

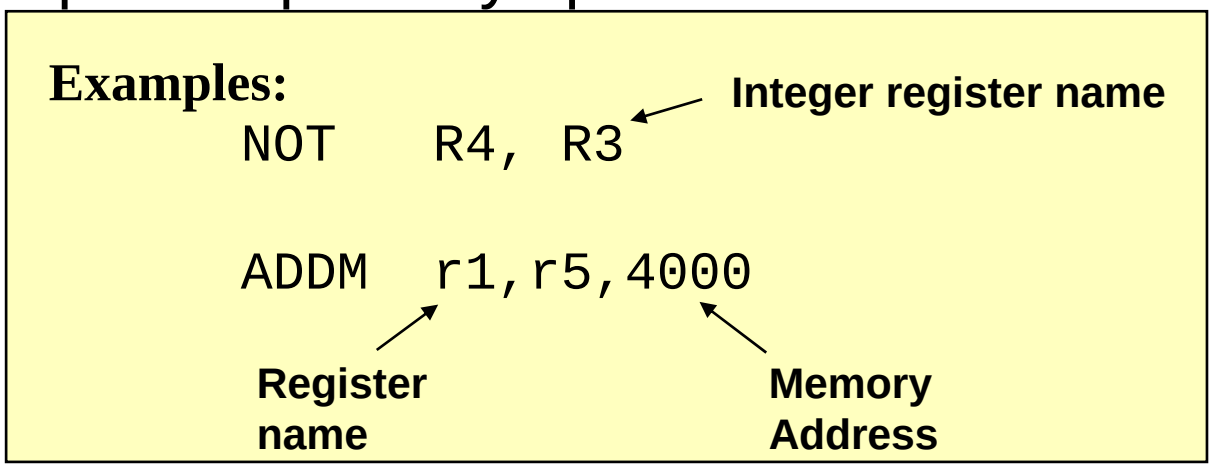
   a) Name (usually explicit) – e.g. 'R13'

Name Spaces

- Each name space $\Rightarrow$

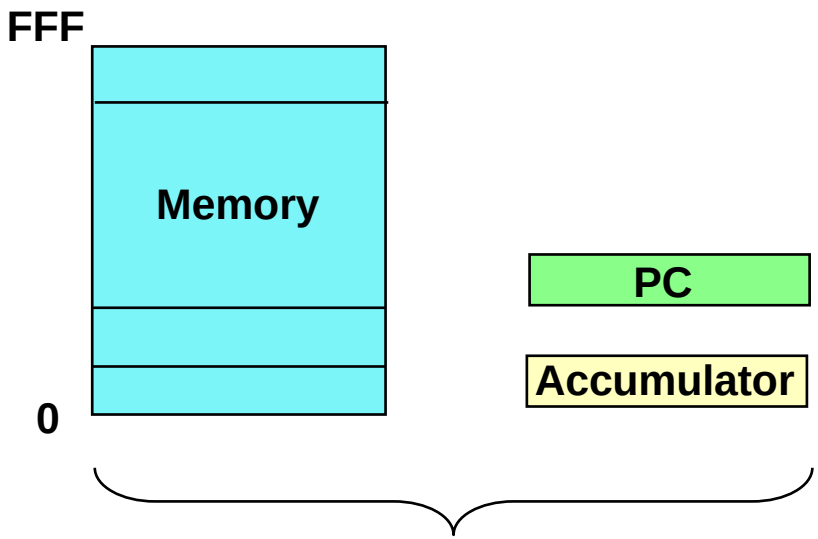
  separately enumerable set of names

- For MIPS there are three namespaces:
  - Integer register numbers
  - Floating point register numbers
  - Memory addresses
- Name space implied by opcode:

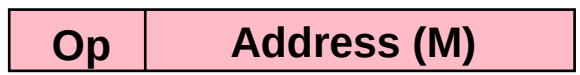


Evolution of Register Organization

- In the beginning…the accumulator
  - 2 instruction types: op and store
    A ← A op M
    A ← A op *M
    *M ← A
  - a one address architecture
    - each instruction encodes one memory address
  - 2 addressing modes
    - *immediate*: M
    - *indirect addressing*: *M
  - Early machines:
    - EDVAC, EDSAC...



Machine State



Instruction Format

(Op encodes addressing mode)

Why Accumulator Architectures?

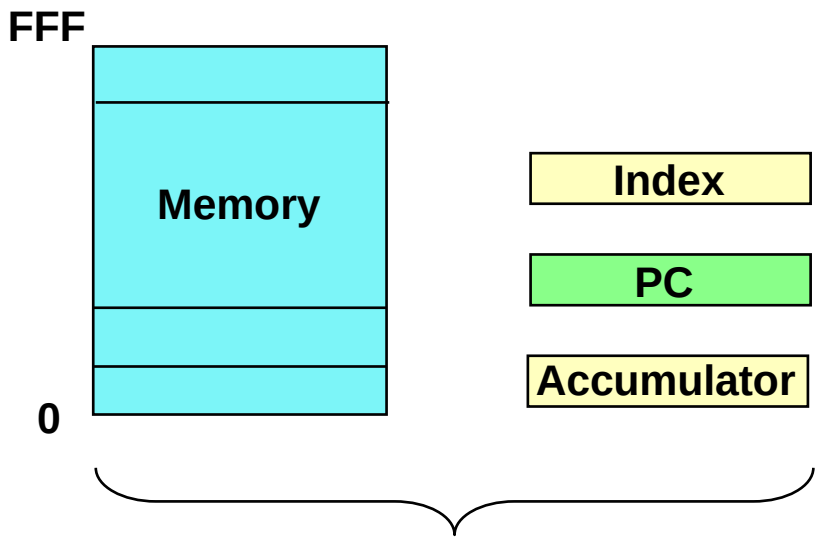
- Registers expensive in early technologies (vacuum tubes)
- Simple instruction decode
  - Logic also expensive
  - Critical programs were small (efficient encoding)
- Less logic $\Rightarrow$ faster cycle time
- Model similar to earlier "tabulating" machines
  - Think adding machine or calculator

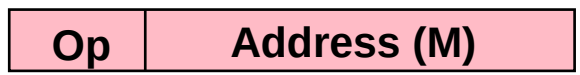
The Index Register

- Add an indexed addressing mode

  A ← A op (M+I)
  A ← A op *(M+I)
  ***(M+I) ← A**

  - good for array access: x[j]
    - address of x[0] in instruction
    - j in index register
  - one register for each key function
    - IP → instructions
    - I → data addresses
    - A → data values
  - new instructions to use I
    - INC I, CMP I, etc.





Instruction Format

Example of Indexed Addressing

```
sum = 0;
for(i=0; i<n; i++)
  sum = sum + y[i];
```

```
START:  CLR     i
        CLR     sum
LOOP:   LOAD    IX
        AND     #MASK
        OR      i
        STORE   IX
        LOAD    sum
IX:     ADD     y
        STORE   sum
        LOAD    i
        ADD     #1
        STORE   i
        CMP     n
        BNE     LOOP
```

Without Index Register

```
START:  CLRA
        CLRX
LOOP:   ADDA    y(X)
        INCX
        CMPX    n
        BNE     LOOP
```

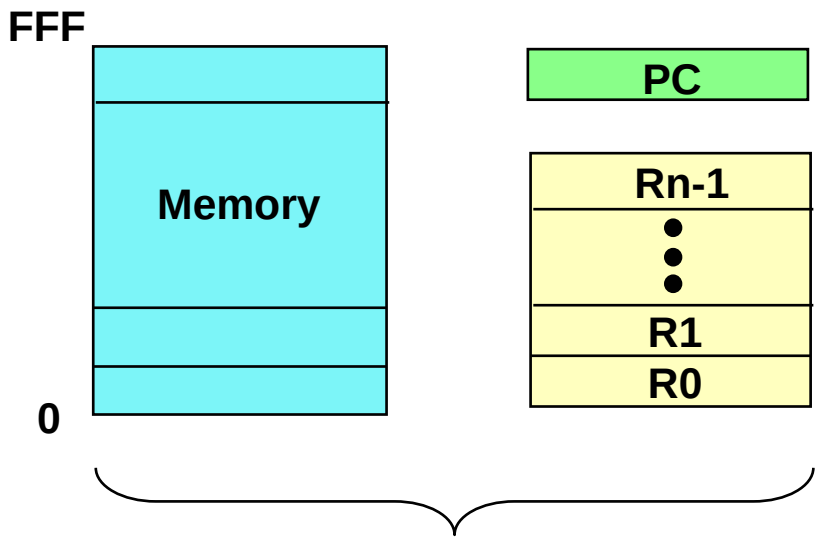
With Index Register

But What About…

```
sum = 0;
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    sum = sum + x[j]*y[i];
```

General Registers

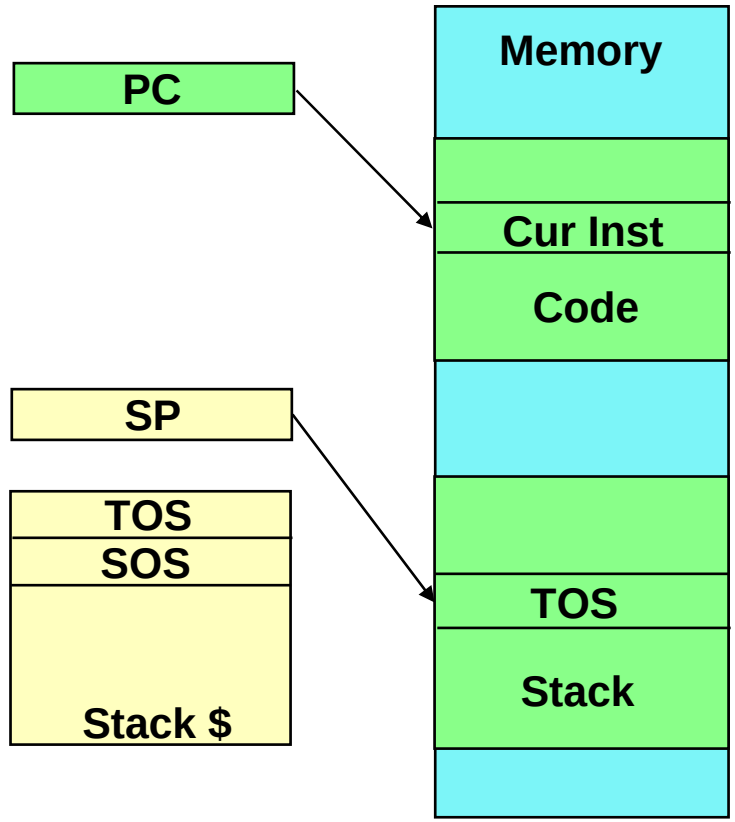
- Merge accumulators (data) and index (address)
- Any register can hold variable or pointer
  - simpler
  - more orthogonal (opcode independent of register usage)
  - More fast local storage
  - but….addresses and data must be same size
- How many registers?
  - More - fewer loads and stores
  - But - more instruction bits



3-address Instruction Format

Stack Machines – like HP's RPN calculators

- Register state is PC and SP
- All instructions performed on TOS (top of stack) and SOS (second on stack)
  - pushes/pops of stack implied
    - op TOS SOS
    - op TOS M
    - op TOS *M
    - op TOS *(M+SP)
- Many instructions are *zero* address
- Stack cache for performance
  - similar to register file
  - hardware managed
- Why do we care? JVM



Examples of Stack Code

```
a = b + c * d;
e = a + f[j] + c;
```

```
PUSH    d
PUSH    c
MUL
PUSH    b
ADD
PUSH    j
PUSHX   f
PUSH    c
ADD
ADD
POP     e
```

Pure Stack
(zero addresses)
11 inst, 7 addr

```
PUSH    d
MUL     c
ADD     b
PUSH    j
PUSHX   f
ADD     c
ADD
POP     e
```

Stack + One Address
8 inst, 7addr

```
LOAD    R1, d
LOAD    R2, c
MUL     R3, R1, R2
LOAD    R4, b
ADD     R5, R4, R3
LOAD    R6, j
LOAD    R7, f(R6)
ADD     R8, R7, R2
ADD     R9, R5, R8
STORE   e, R9
```

Load/Store
(many GP registers)
10 inst, 6addr

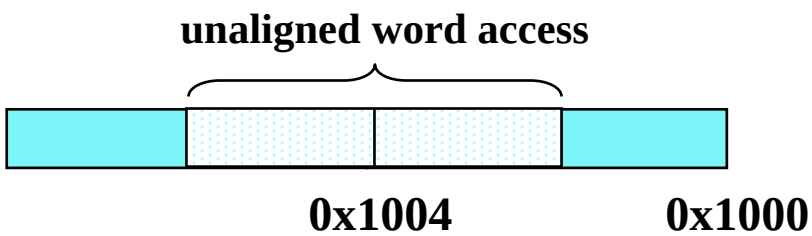
Memory Organization

Memory Organization

- Four components specified by ISA:
  - Smallest addressable unit of memory (byte? halfword? word?)
  - Maximum addressable units of memory (doubleword?)
  - Alignment
  - Endianness
- Already talked about addressing modes last time

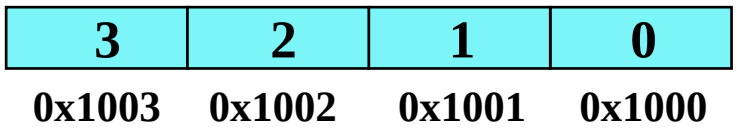
Alignment

- Some architectures restrict addresses that can be used for particular size data transfers!
  - Bytes accessed at any address
  - Halfwords only at even addresses
  - Words accessed only at multiples of 4

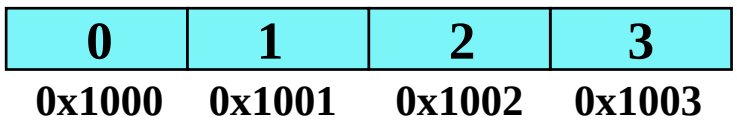


Endianness

- How are bytes ordered within a word?
  - Little Endian (Intel/DEC)



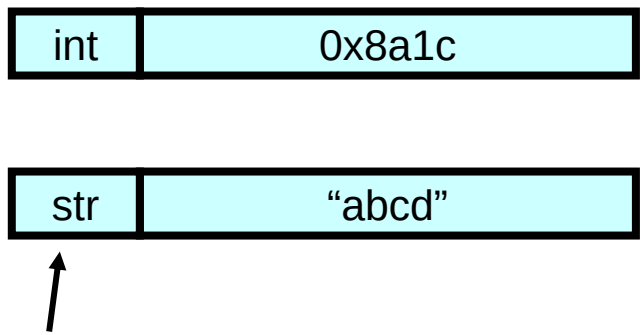
  - Big Endian (IBM/Motorola)



  - Today - most machines can do either (configuration register)

Data Types

- How the contents of memory and registers are interpreted
- Can be identified by
  - Tag (data itself)
  - Use (instructions)
- Driven by application
  - Signal processing
    - 16-bit fixed point (fraction)
  - Text processing
    - 8-bit characters
  - Scientific computing
    - 64-bit floating point
- Most *general purpose* computers support several types
  - 8, 16, 32, 64-bit
  - signed and unsigned
  - fixed and floating



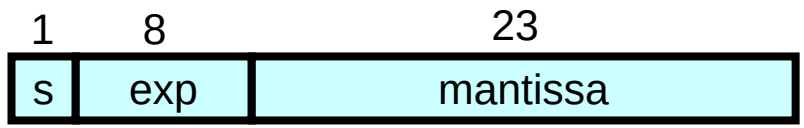
Examples of tags (ie. Symbolics machine)

Example: 32-bit Floating Point

- Type specifies mapping from bits to real numbers (plus symbols)
  - format
    - S, 8-bit exp, 23-bit mantissa
  - interpretation
    - mapping from bits to abstract set

$$v = (-1)^S \times 2^{(E-127)} \times 1.M$$

  - operations
    - add, mult, sub, sqrt, div



Summary

- Summary
  - Register organization
    - Trading instruction density for flexibility
  - Various vagaries of memory instructions
    - Alignment, endian

- Next Time – Steve Keckler guest lecture
  - Reading assignment – P&H B.1-B.3
  - Exceptions
  - Wrap-up ISA issues
  - Logic design of microarchitectural elements

- Architecture seminar: Monday 3:30 ACES 2.302