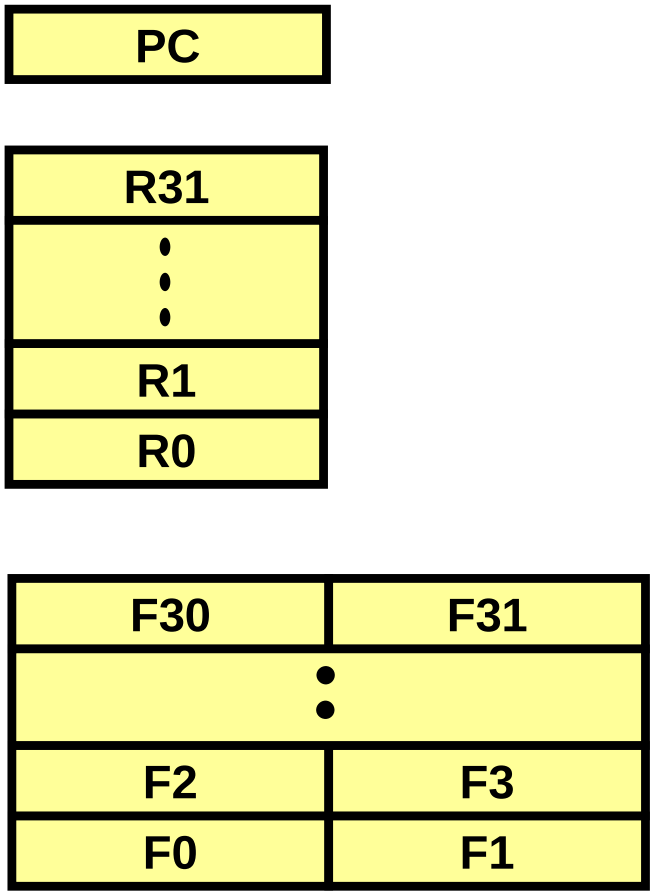
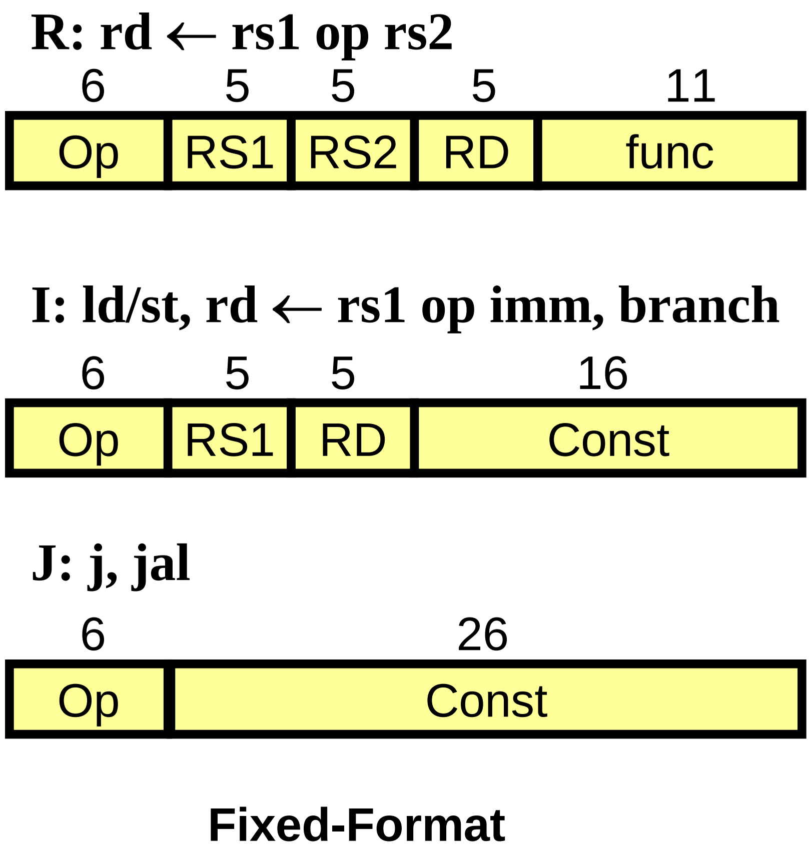
Lecture 6: Instruction Set Architectures - III

- Announcements:
  - Readings for today: P&H 2.7-2.19
- Last Time
  - MIPS ISA and discussion
- Today
  - Quick review
  - Case study #2: graphics processor ISA
  - Register organization
  - Memory addressing

Last time - MIPS ISA (a visual)





MIPS Instruction Types

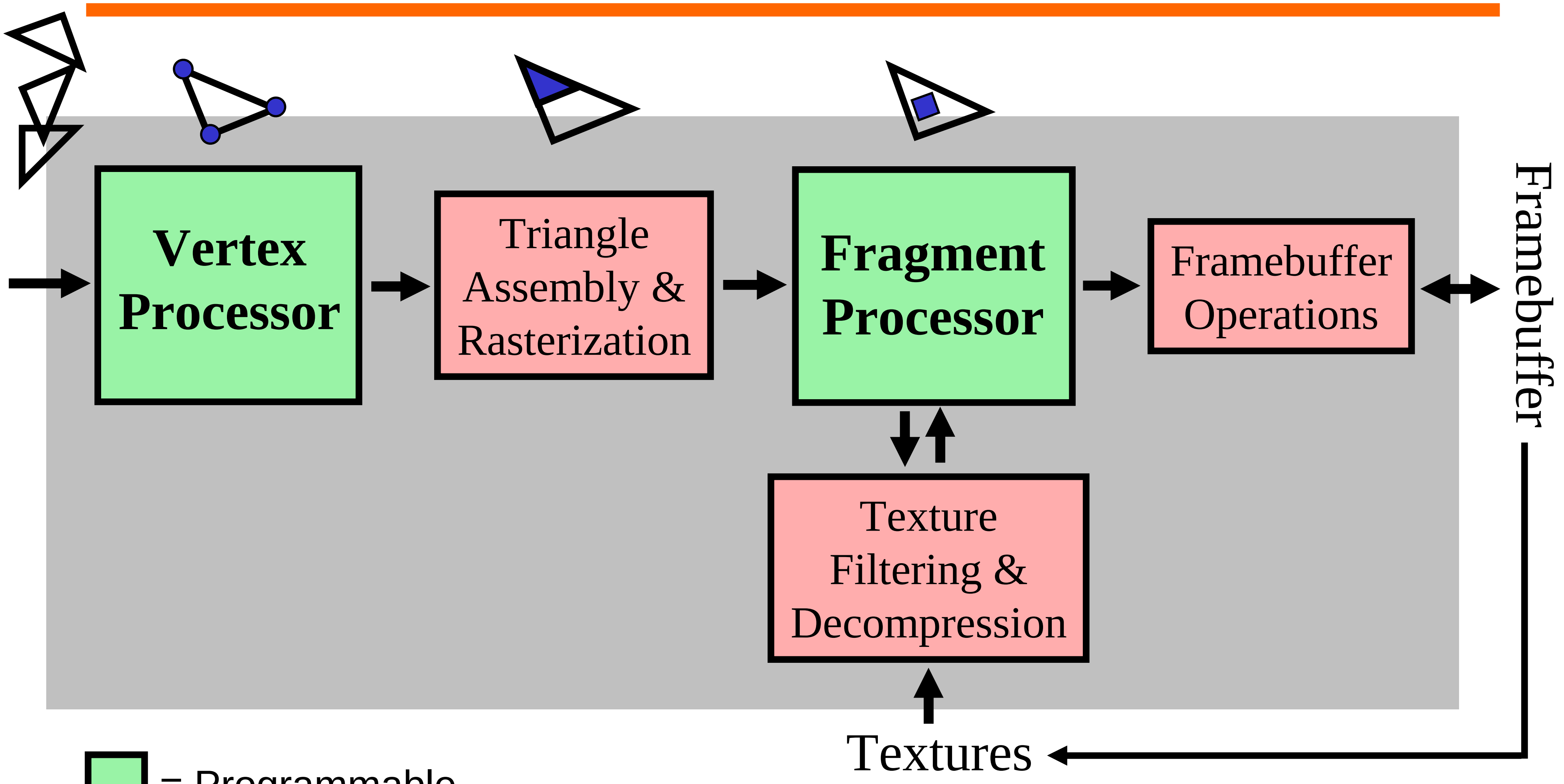
- **ALU Operations**
  - arithmetic – int and float (`add, sub, mult`)
  - logical (`and, or, xor, srl, sra`)
  - data type conversions (`cvt.w.d, cvt.s.d`)
- **Data Movement**
  - memory reference (`lb, lw, sb, sw`)
  - register to register (`move, mfhi`)
- **Control** - what instruction to do next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

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 into the link register (R31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
- It cannot occur in
  - addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

ISA for a modern graphics processor

A modern graphics processor

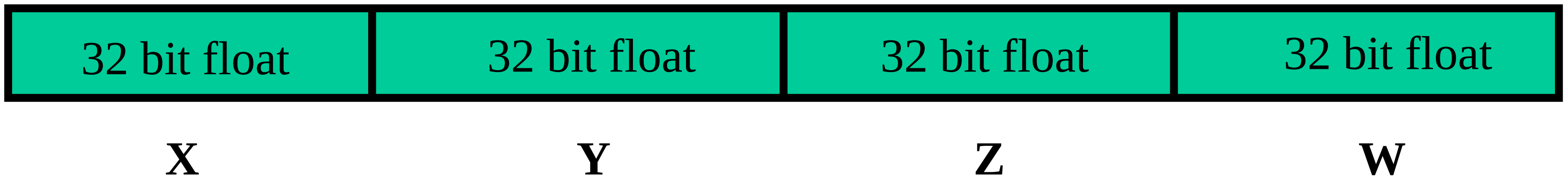


🟩 = Programmable

🟥 = Not programmable – hardwired algorithms

A GPU Register

128 bits wide



- Instructions can access and rearrange individual components:

ADD R0, R1.wzxy, R2.xxww →

R0.x = R1.w + R2.x
R0.y = R1.z + R2.x
R0.z = R1.x + R2.w
R0.w = R1.y + R2.w

- Lots of 4-vector computations in 3D graphics

Vertex Processor Instructions

- 4-vector arithmetic – add, multiply, and combinations
  - ADD R0, R1.wzxy, R2.xyzw
- Scalar arithmetic – reciprocal, square root, etc..
  - RSQ R3.x, R4.x
- Specialized arithmetic
  - LIT R0, R1
- Control flow
  - BRA target, (EQ.x)
- Move
- Misc – including pack/unpack, and data conversion

Condition codes and predication

- Condition codes may be set on any operation
  - By appending "C" to opcode: e.g. ADDC instead of ADD

- There are four sets of condition codes (for x,y,z,w)

- Set to indicate:
  - Less than zero
  - Equal to zero
  - Greater than zero
  - Unordered (e.g. NaN)

- Branches can use one condition code
- Other instructions can be predicated

Example: MOV R1.xy (NE.z), R0;

- Copy R0 components to R1's X & Y components
- *except* when condition code's Z component is EQ

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:

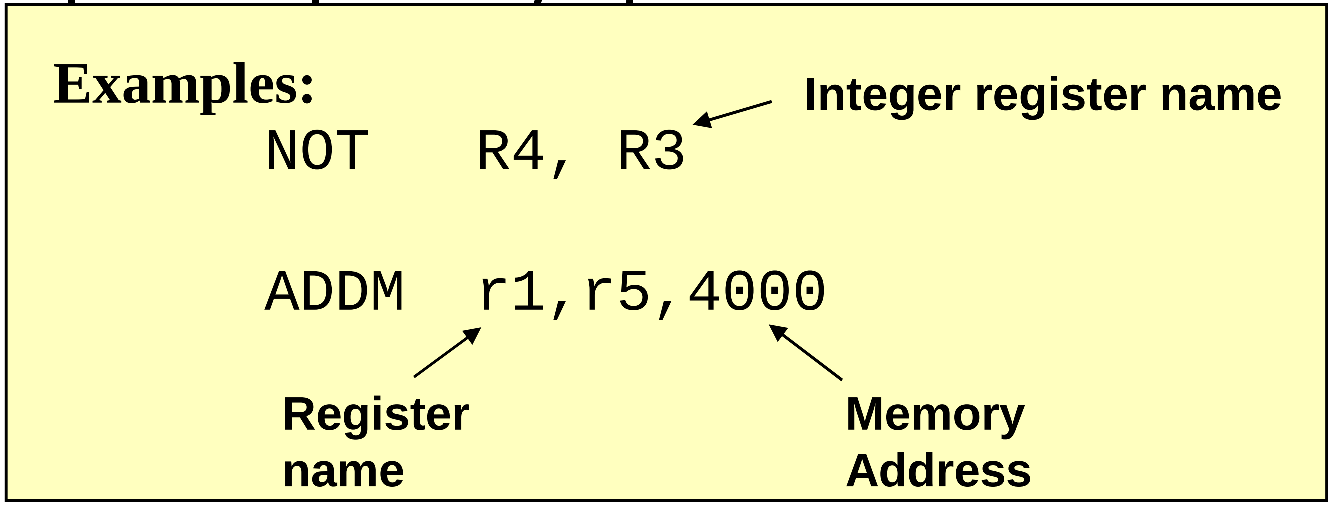
b) Namespace (usually implicit) – e.g 'registers'

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

Name Spaces

- Each name space ⇒

  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:

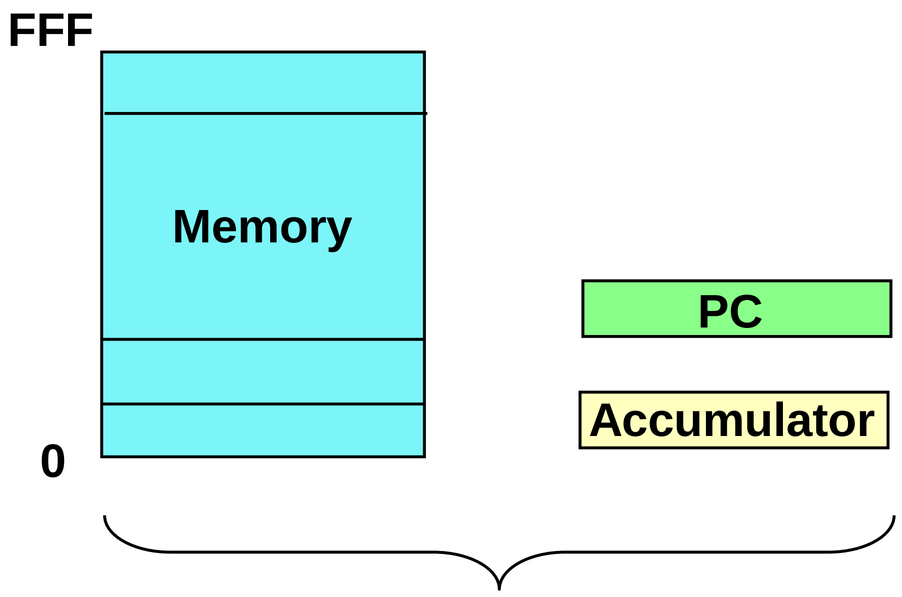


Examples:

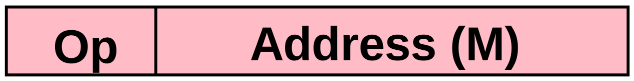
```
NOT   R4, R3

ADDM  r1,r5,4000
```

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

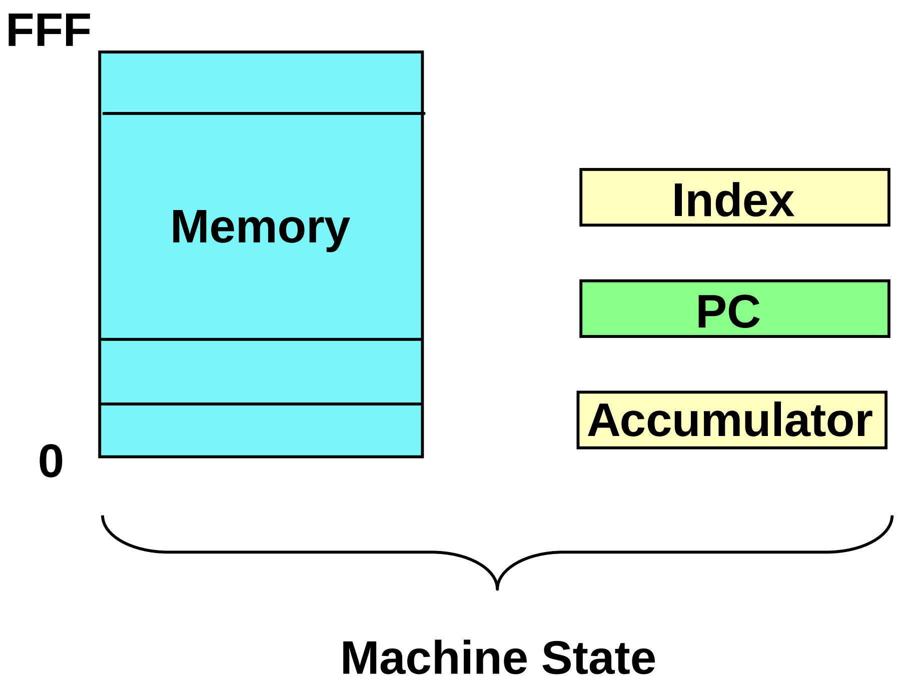
(Op encodes addressing mode)

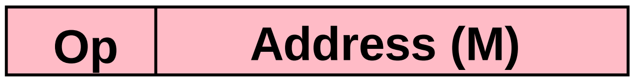
Why Accumulator Architectures?

- 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
  - **$A \leftarrow A$ op $(M+I)$**
  - **$A \leftarrow A$ op $*(M+I)$**
  - **$*(M+I) \leftarrow A$**
  - good for array access: x[j]
    - address of x[0] in instruction
    - j in index register
  - one register for each key function
    - IP $\rightarrow$ instructions
    - I $\rightarrow$ data addresses
    - A $\rightarrow$ data values
  - new instructions to use I
    - INC I, CMP I, etc.

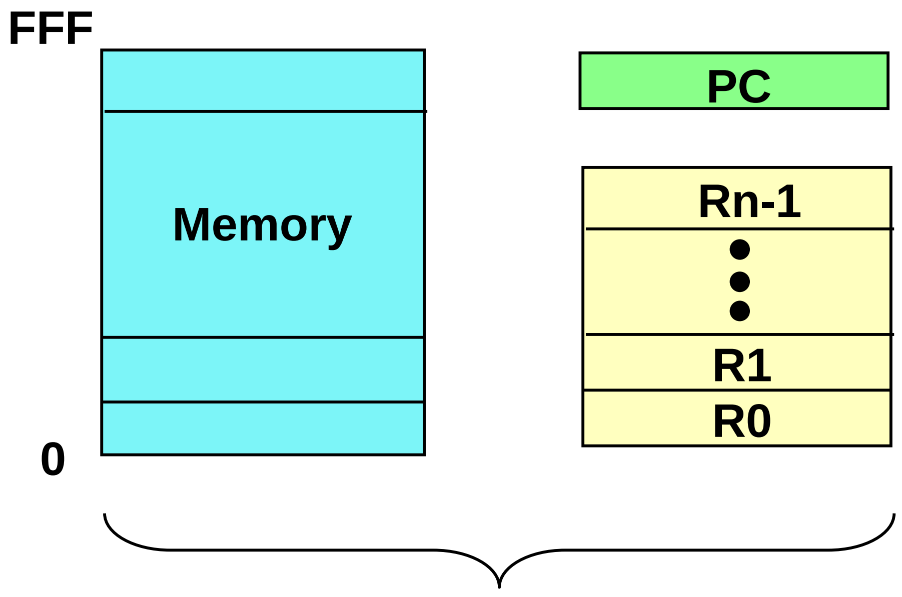




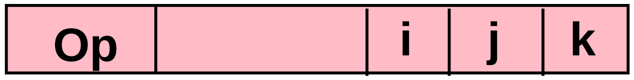
Instruction Format

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



Machine State

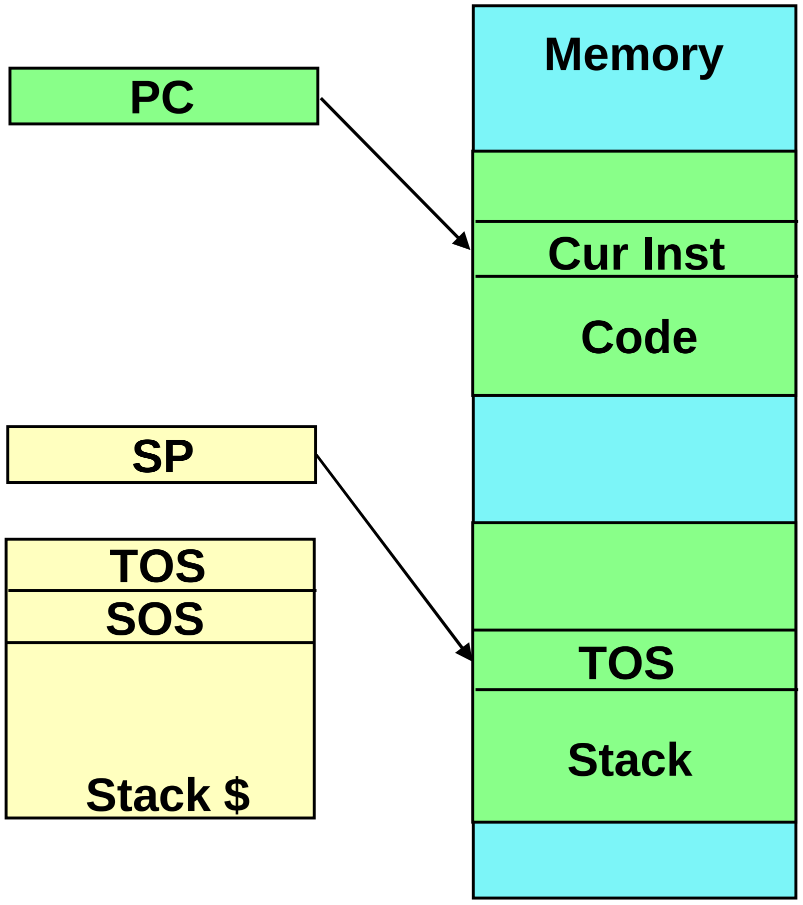


3-address Instruction Format

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

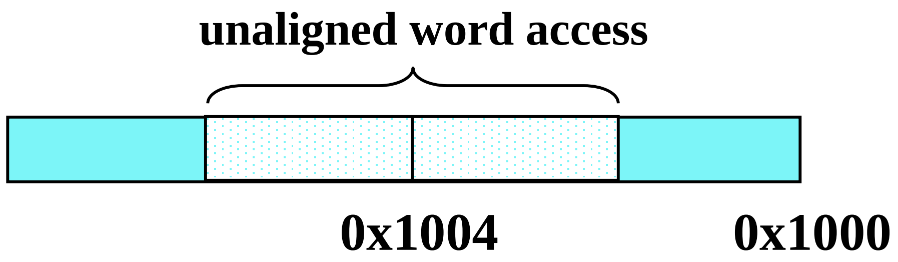
Memory Organization

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

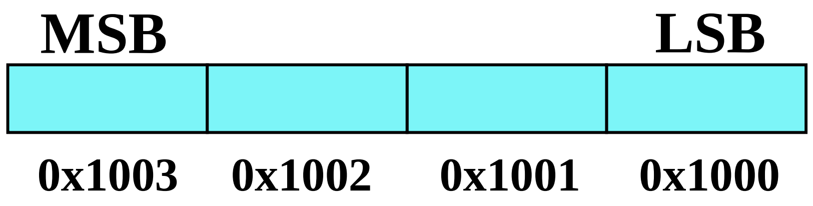
Alignment

- 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

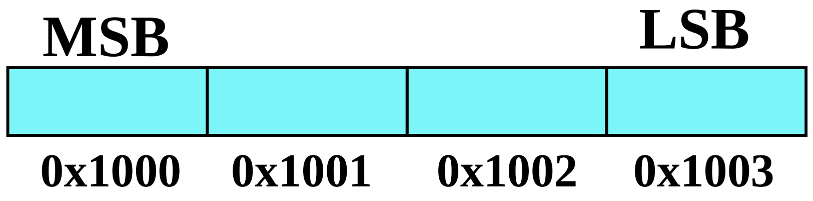


Endianness

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



  - Big Endian (IBM/Motorola)



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

Summary

- ISA principles
- Graphics processor ISA

- Next Time
  - Data Types, begin pipelinling
  - Reading assignment – 3.2 (int), 3.6 (float), B.9 (memory)