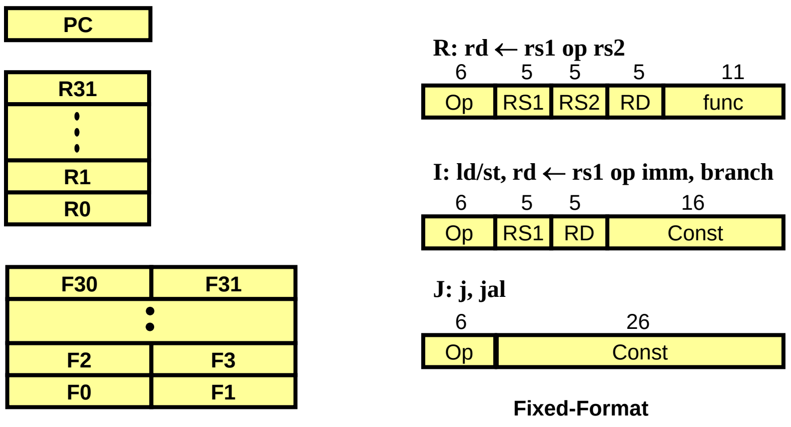
Lecture 7: Instruction Set Architectures - IV

- Announcements:
  - Readings for today: 3.2 (int), 3.6 (float), B.9 (memory)
- Last Time
  - MIPS ISA and discussion
- Today
  - Case study #2: graphics processor ISA
  - Register organization
  - Memory addressing

Last time - MIPS ISA (a visual)

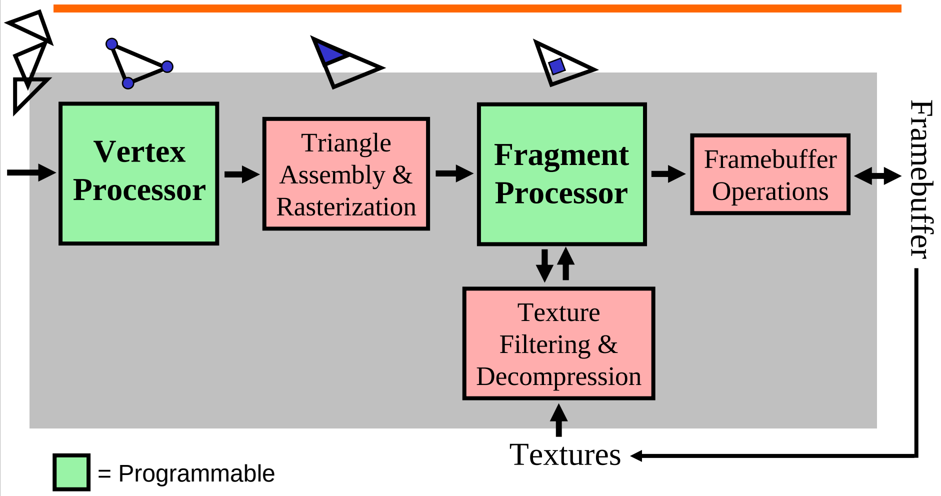


MIPS Instruction Types

- ***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`)

ISA for a modern graphics processor

A modern graphics processor



🟩 = Programmable

🟥 = Not programmable – hardwired algorithms

A GPU Register

128 bits wide

32 bit float	32 bit float	32 bit float	32 bit float
X	**Y**	**Z**	**W**

- 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

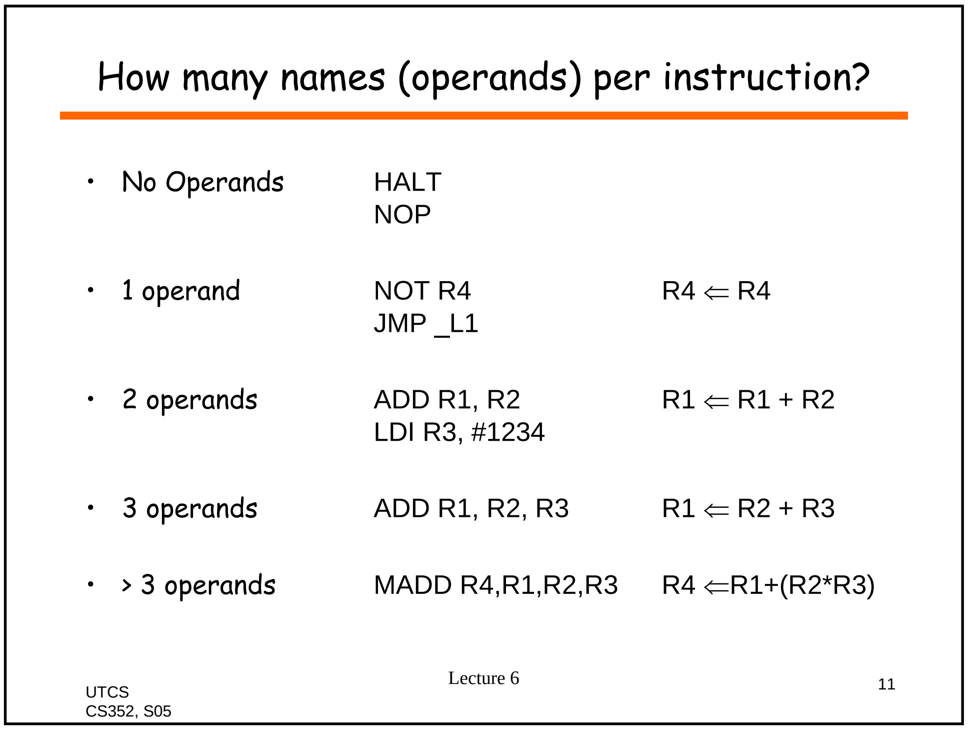
- Condition codes may be set on any operation
  - By appending "C" to opcode: e.g. ADDC instead of ADD
- There are four sets of condition codes (for x,y,z,w)
- Set to indicate:
  - Less than zero
  - Equal to zero
  - Greater than zero
  - Unordered (e.g. NaN)
- Branches can use one condition code
- 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

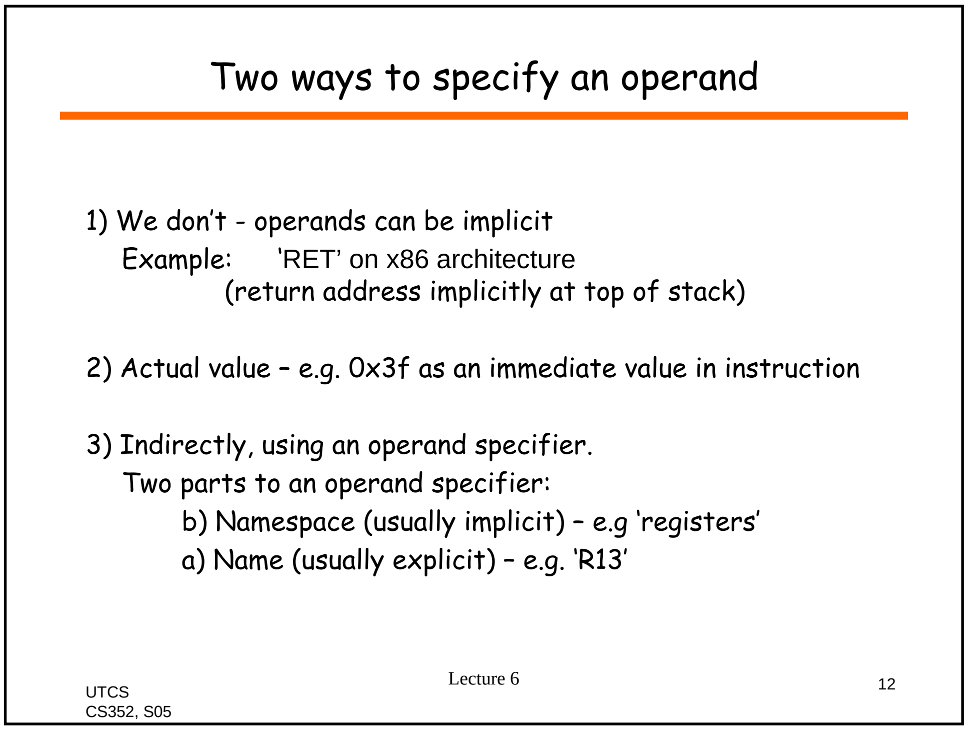
Naming Storage in ISAs

- 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
```

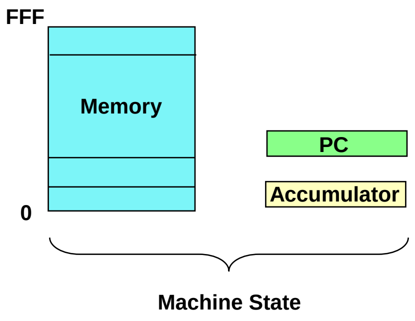
Integer register name

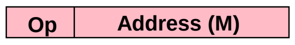
Register name

Memory Address

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...





Instruction Format

(Op encodes addressing mode)

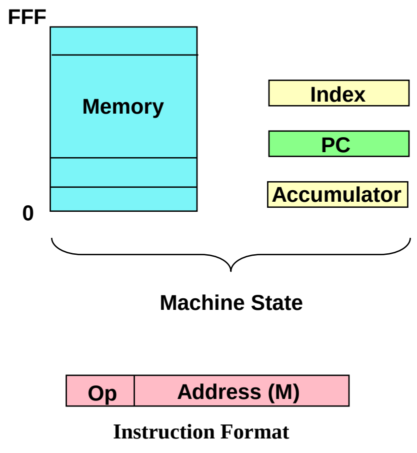
Why Accumulator Architectures?

- 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 <u>adding machine</u> or <u>calculator</u>

The Index Register

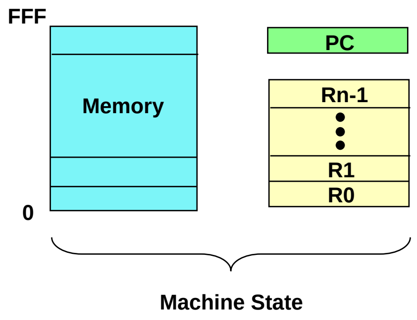
- Add an indexed addressing mode

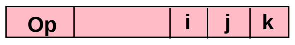
  A ← A op (M+I)
  A ← A op *(M+I)
  ***(M+I) ← A**
  - good for array access: x[j]
    - address of x[0] in instruction
    - j in index register
  - one register for each key function
    - IP → instructions
    - I → data addresses
    - A → data values
  - new instructions to use I
    - INC I, CMP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



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

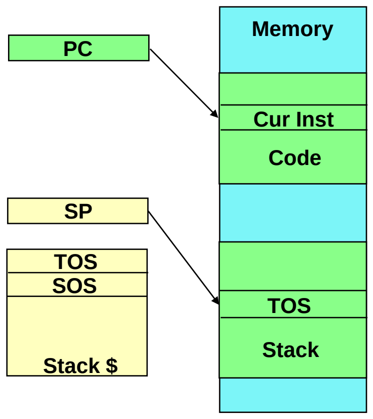




3-address Instruction Format

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

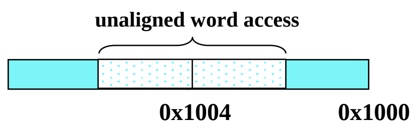
Memory Organization

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

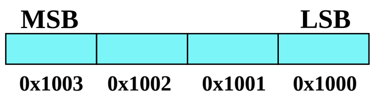
Alignment

- 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

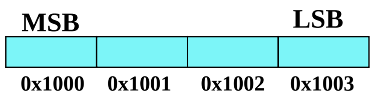


Endianness

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



  - Big Endian (IBM/Motorola)

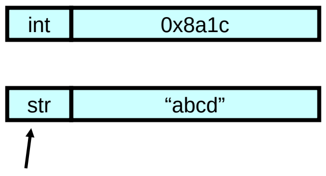


  - Today - most machines can do either (configuration register)

Data Types

Data Types

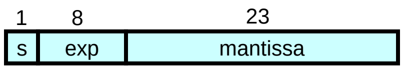
- How the contents of memory and registers are interpreted
- Can be identified by
  - Tag (data itself)
  - Use (instructions)
- Driven by application
  - Signal processing
    - 16-bit fixed point (fraction)
  - Text processing
    - 8-bit characters
  - Scientific computing
    - 64-bit floating point
- 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



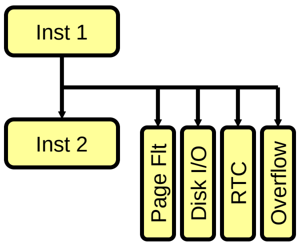
$$v = (-1)^S \times 2^{(E-127)} \times 1.M$$

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

Exceptions

Control - Exceptions/Events

- Implied branch after every instruction
  - Internal events
    (called faults or exceptions)
    - arithmetic overflow
    - page fault
  - External events
    (called interrupts)
    - completion of I/O operations



- What happens????
  - Save PC (in special register)
  - Jump to exception address
    (taken from table)
  - When done: Jump to original PC + 4

General ISA design principles

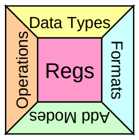
UTCS
CS352, S05

Lecture 6

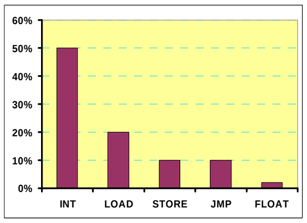
29

Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers



- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others



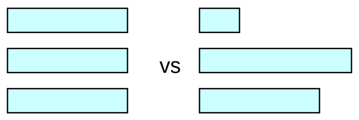
UTCS
CS352, S05

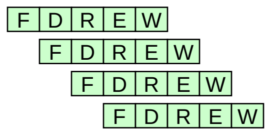
Lecture 6

30

Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features/instructions
  - principle of *least surprise*
  - performance should be easy to predict
- Cost effectiveness
  - design ISA to permit efficient implementation
    - today
    - 10 years from now





CISC vs RISC

- What is a RISC?
  (Reduced Instruction Set Computer)
  - no firm definition
  - generally includes
    - general registers
    - fixed 3-address instruction format
    - strict load-store architecture
    - simple addressing modes
    - simple instructions
  - Examples
    - DEC Alpha
    - MIPS
  - Advantages
    - good compiler target
    - easy to implement/pipeline
- CISC (Complex Instruction-Set Computer)
  - CISC ≡ ¬RISC
  - may include
    - variable length instructions
    - memory-register instructions
    - complex addressing modes
    - complex instructions
      - CALLP, EDIT, …
  - Examples
    - DEC VAX, IBM 370, x86
  - Advantages
    - better code density
    - legacy software

Compilers

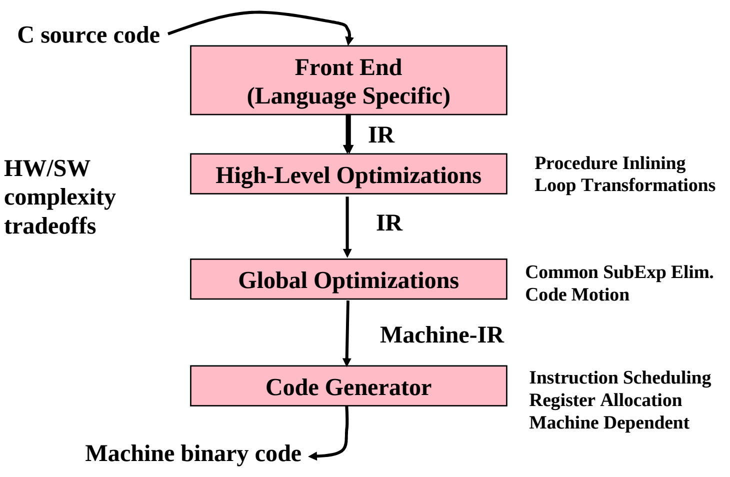
Performance = application +
compiler +
hardware

UTCS
CS352, S05

Lecture 6

33

Role of the Optimizing Compiler



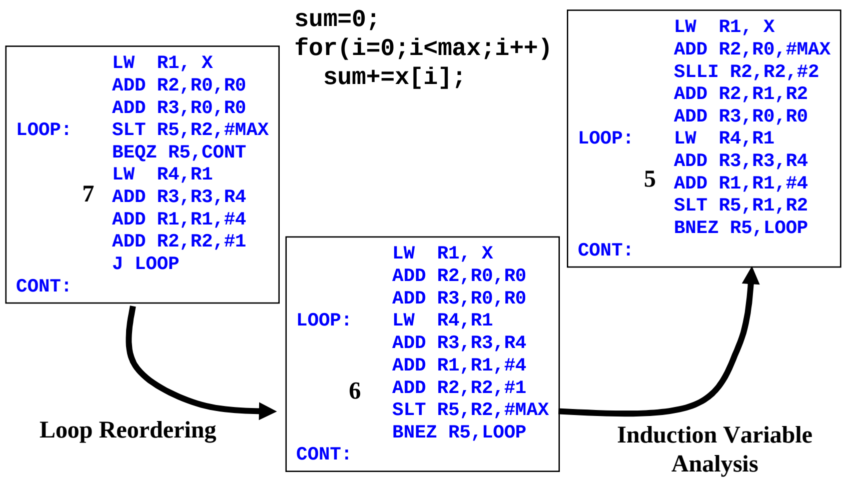
UTCS
CS352, S05

Lecture 6

34

Example: Loop Optimization

```
sum=0;
for(i=0;i<max;i++)
  sum+=x[i];
```



```
         LW   R1, X
         ADD  R2,R0,R0
         ADD  R3,R0,R0
LOOP:    SLT  R5,R2,#MAX
         BEQZ R5,CONT
         LW   R4,R1
      7  ADD  R3,R3,R4
         ADD  R1,R1,#4
         ADD  R2,R2,#1
         J LOOP
CONT:
```

```
         LW   R1, X
         ADD  R2,R0,R0
         ADD  R3,R0,R0
LOOP:    LW   R4,R1
         ADD  R3,R3,R4
         ADD  R1,R1,#4
      6  ADD  R2,R2,#1
         SLT  R5,R2,#MAX
         BNEZ R5,LOOP
CONT:
```

```
         LW   R1, X
         ADD  R2,R0,#MAX
         SLLI R2,R2,#2
         ADD  R2,R1,R2
         ADD  R3,R0,R0
LOOP:    LW   R4,R1
         ADD  R3,R3,R4
      5  ADD  R1,R1,#4
         SLT  R5,R1,R2
         BNEZ R5,LOOP
CONT:
```

Architect can help Compiler Writer

- Simplify, Simplify, Simplify
  - Feature difficult to use, it won't be used….Less is More!
- Regularity
  - Common set of formats, few special cases
- Primitive, not solutions
  - CALLS vs. Fast register moves
- Make performance tradeoffs simple
- Ultimately, the ISA will *not* be perfect

Compiler can help Microarchitecture

- Instruction Scheduling
  - Instruction Level Parallelism
- Resource Allocation
  - Registers (minimize spills/restores to and from memory)
- Memory optimizations
  - Cache conscious data organization
  - Code layout
- Etc……

Summary

- ISA principles
- Compiler/ISA interaction
- Next Time
  - Simple processor datapath
  - Reading assignment – P&H 5.1-5.4