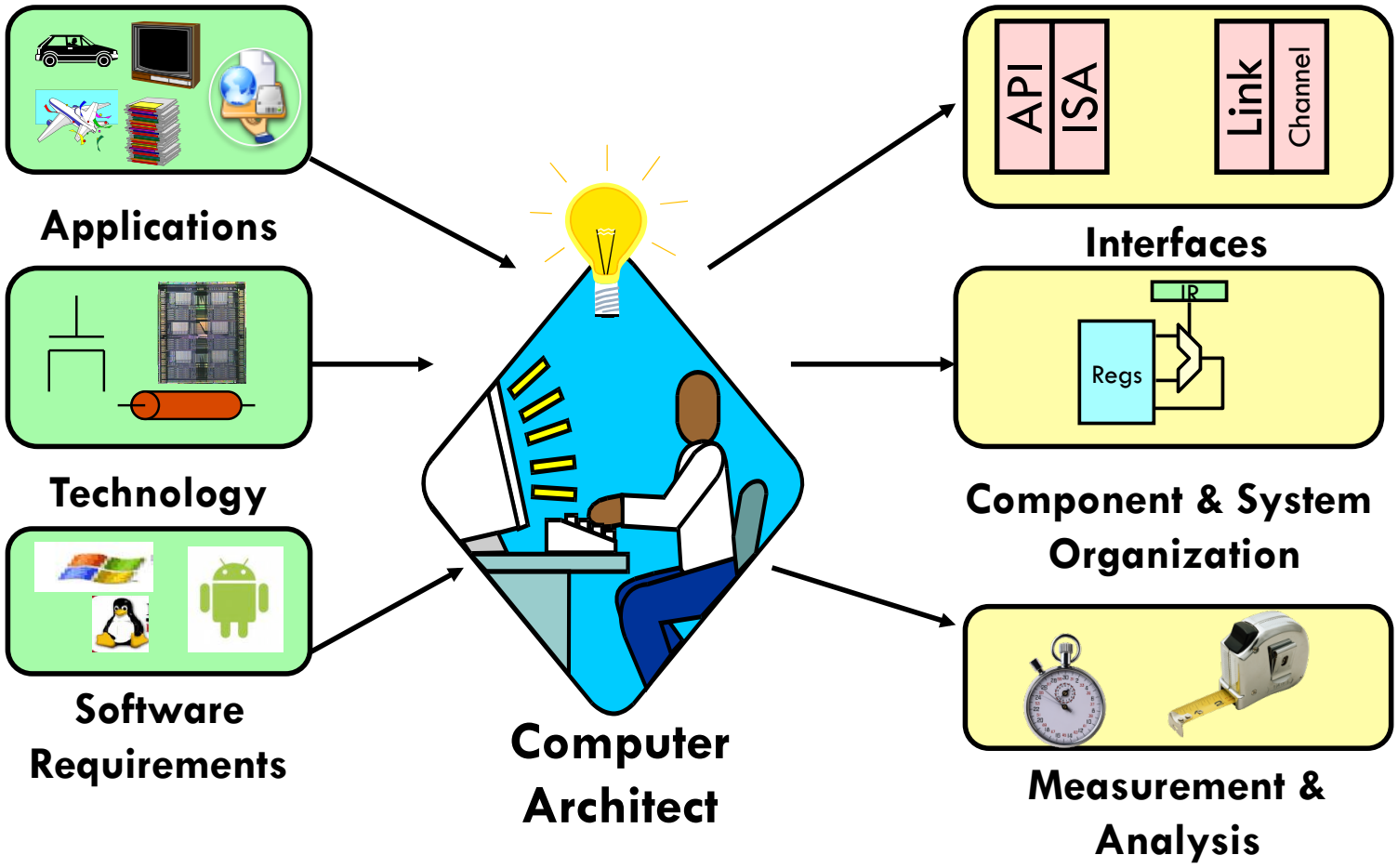
Instruction Set Architecture (ISA) (1)

Lecture 2a
September 6th, 2023

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Instructor's slides from Elsevier Inc.*

Review - What Computer Architects Do?



Source: Stanford EE282 (Prof. C. Kozyrakis)

Outline

Textbook: Appendix A (Instruction Set Principles)

- **What is ISA?**
- **ISA Basics – What ISA defines?**
  - Register Organization
  - Memory Organization
  - Addressing Modes
  - Instruction Types
  - Instruction Formats
  - ISA Principles

ISA Analogy – Interface vs. Implementation

- **Separation of interface from implementation**
  - Example: printf() function call in C

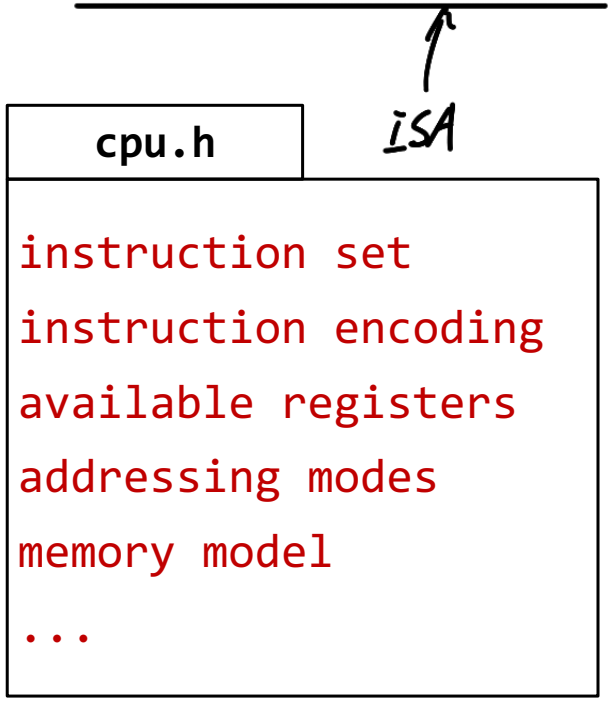
```
// In your code helloworld.c:
#include <stdio.h>
int main() {  printf ("Hello World!\n");  }
```

```
// In /usr/include/stdio.h:
// This file defines the interface of printf()
extern int printf (__const char *__restrict __format, ...);
```

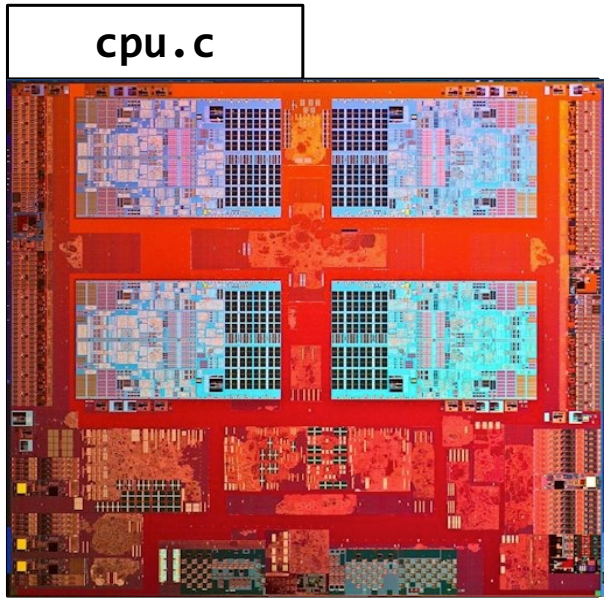
```
// In ../glibc/stdio-common/printf.c:
// This file contains the actual implementation of printf()
int __printf (const char *format, ...)
{ ... }
```

ISA Analogy – Interface vs. Implementation

- **What if we apply the same idea to CPU?**
  - Separating interface (for users) from implementation (for designers)
  - Interface -> **architecture** / implementation -> **microarchitecture**



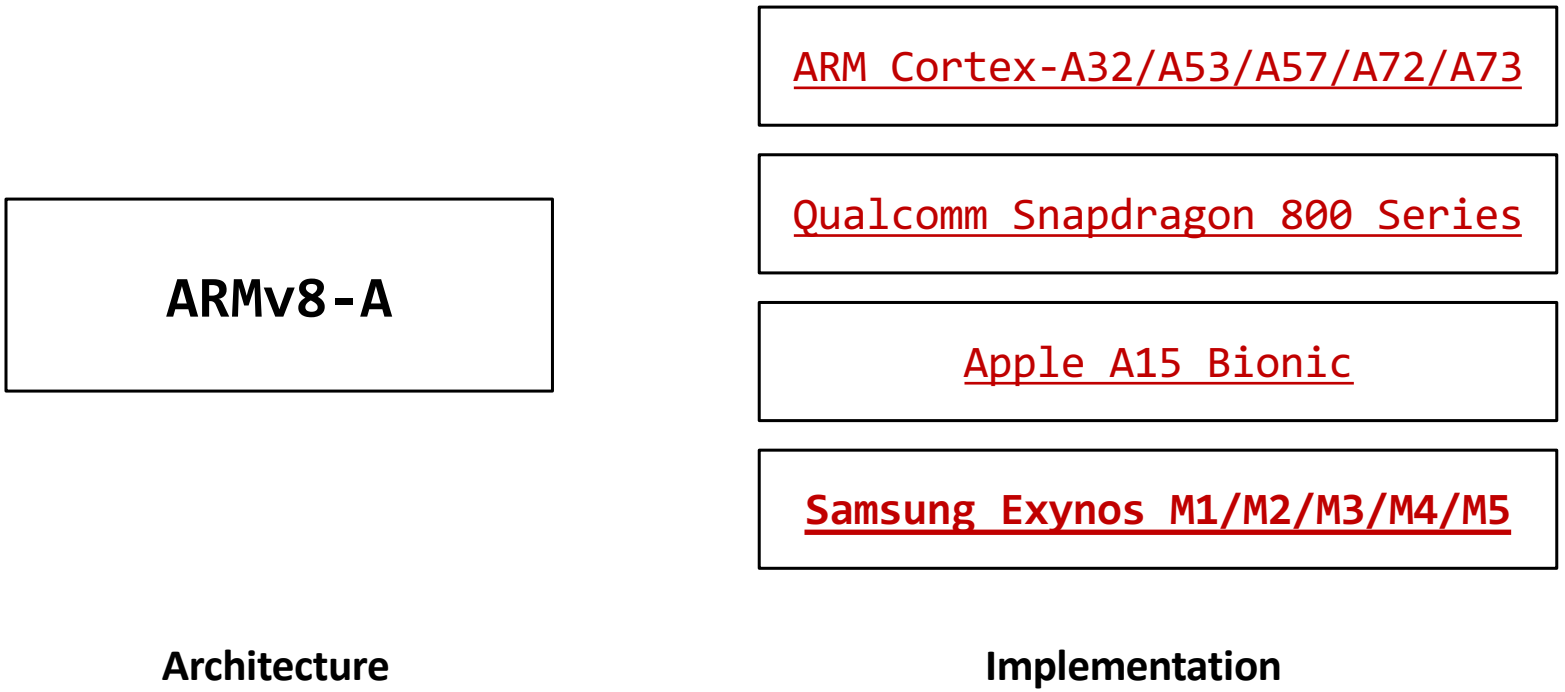
Architecture



Implementation

ISA Analogy – Interface vs. Implementation

- **There may be many microarchitectures implementing the same architecture.**
  - Possibly from multiple vendors



ISA Analogy – Interface vs. Implementation

- **Examples: Architecture or Implementation?**
  - Number of software-visible general-purpose registers A
  - Width of memory bus I (not software-visible)
  - Binary representation of the instruction `sub r4,r2,#27` A
  - Number of cycles to execute floating-point (FP) instruction I (performance problem)
  - How condition code bits are set on a move instruction A
  - Size of the instruction cache I
  - Type of FP format A

register-renaming (more physical registers)

role of ISA definition

ISA Analogy – Interface vs. Implementation

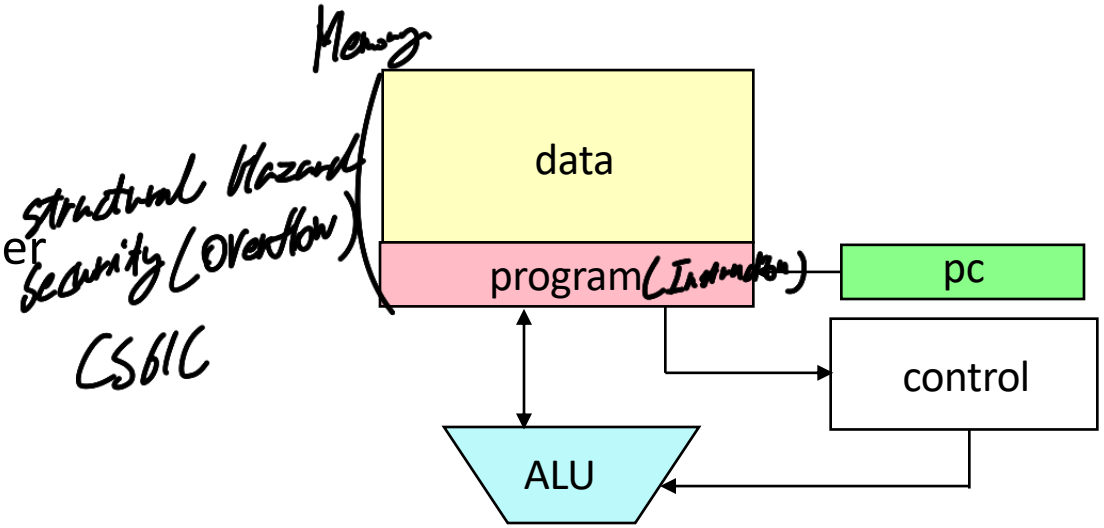
- **Intel64 ISA book** x86-64
  - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- **ARMv8 (64-bit) ISA book**
  - https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
- **RISC-V ISA book** (recent)
  - https://riscv.org/technical/specifications/

What is ISA?

- **Contract between programmer and the hardware**
  - Defines visible state of the system
  - Defines how state changes in response to instructions
- **Programmer: ISA is model of how a program will execute**
- **Hardware Designer: ISA is formal definition of the correct way to execute a program**
- **ISA specification**
  - The binary encodings of the instruction set

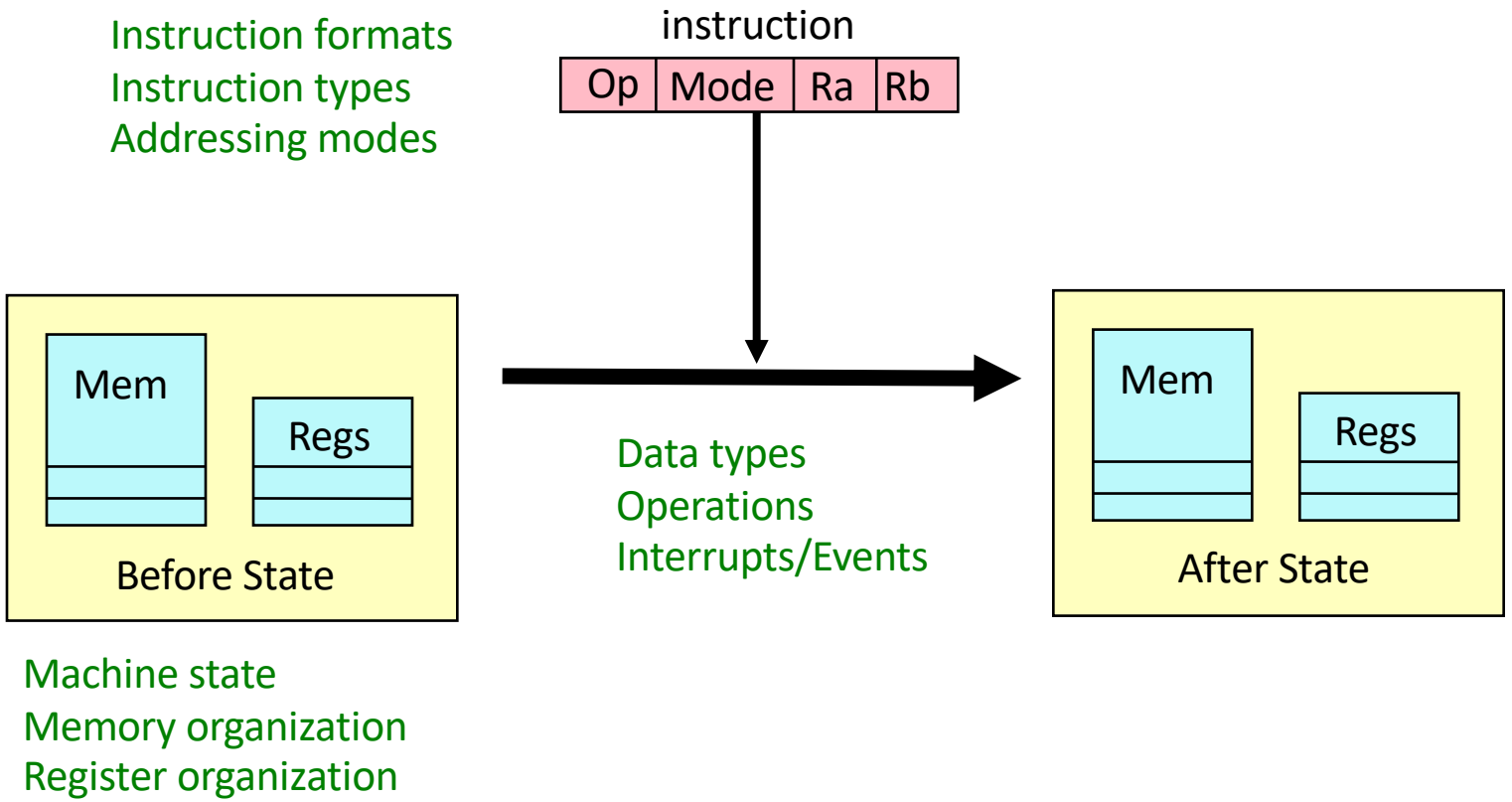
What is ISA?

- **Modern ISAs are based on stored program computer model**
  - The "von-Neumann" memo: "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument"
    - Program = A sequence of instructions
    - The same storage can be used to store both program and data
  - Storage
    - Program
    - Data
  - Program counter
  - ALU



What is ISA?

- **What does a typical ISA define?**



What is ISA?

- **Architecture vs. Implementation**
  - (Instruction Set) Architecture: defines what a computer system does in response to a program and a set of data
    - Programmer visible elements of computer system
  - Implementation (or microarchitecture): defines how a computer does it
    - Sequence of steps to complete operations
    - Time to execute each operation
    - Hidden "bookkeeping" functions
    - e.g., x86_64 microarchitectures (Intel's Skylake, Ice Lake, Sapphire Rapids, Emerald Rapids, …)

Topic of this week

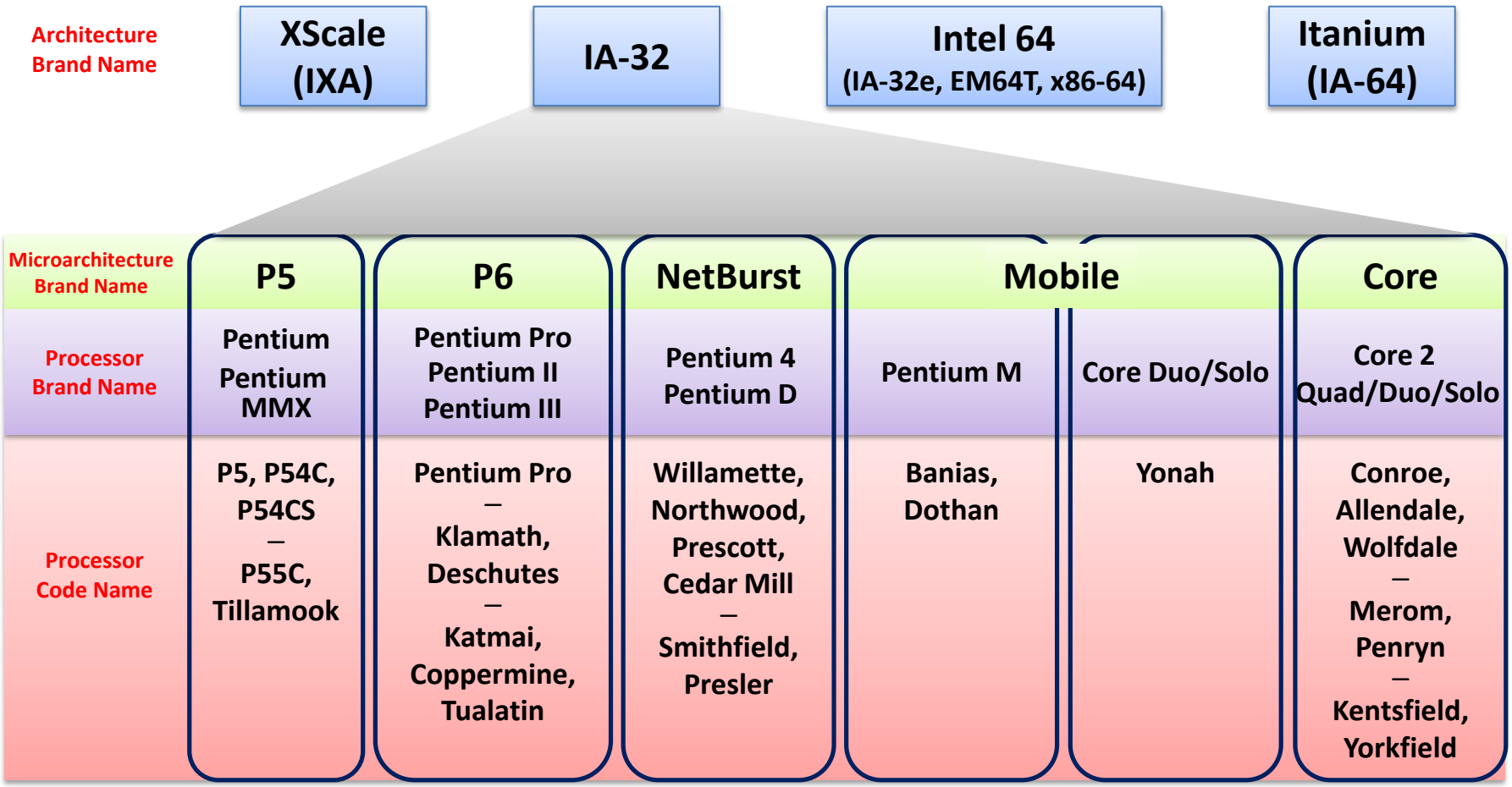
Topic of next 2-3 weeks

What is ISA?

- **Why Separate ISA from Implementation?**
  - Compatibility : Abstraction is gift of god.
    - VAX architecture: mainframe ⇒ single chip
    - ARM: 20x performance range
      - high vs. low performance, power, price
  - Longevity (hard to migrate to new ISA generation)
    - 20+ years of ISA
    - e.g., x86 in 13th generation of Intel® Core™ processors (Raptor Lake)
      - enhancements at generation boundaries
    - retain software investment
    - amortize development costs over multiple markets

What is ISA?

- Architecture vs. Microarchitecture: Intel's IA-32



What is ISA?

- **Architecture vs. Microarchitecture: Intel's x86_64 uArch**

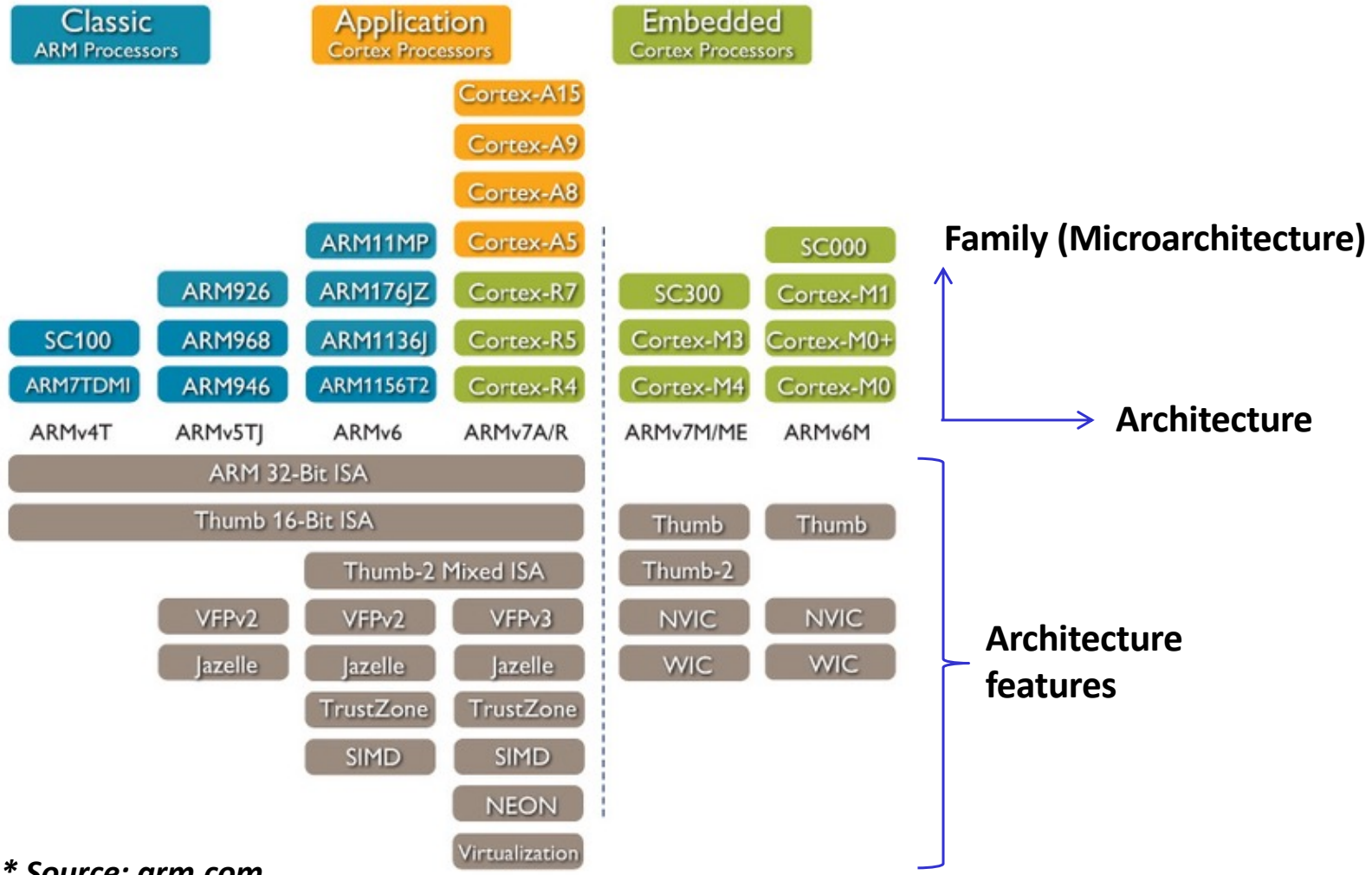
Intel x86_64 Microarchitectures		
µarch	Introduction	Process
Core	2006-04-01	65 nm
Montecito	2006-07-18	90 nm
Polaris	2007-02-01	65 nm
Montvale	2007-10-31	90 nm
Penryn	2007-11-01	45 nm
Bonnell	2008-03-02	45 nm
Nehalem	2008-08-01	45 nm
Rock Creek	2009-12-01	45 nm
Westmere	2010-01-01	32 nm
Tukwila	2010-02-08	65 nm
Knights Ferry	2010-05-31	45 nm
Sandy Bridge (client)	2010-09-13	32 nm
Saltwell	2011-01-01	32 nm
Knights Corner	2011-01-01	22 nm
Ivy Bridge	2011-05-04	22 nm
Poulson	2012-11-08	32 nm
Silvermont	2013-01-01	22 nm
Haswell	2013-06-04	22 nm
Broadwell	2014-10-01	14 nm
Airmont	2015-01-01	14 nm
Skylake (client)	2015-08-05	14 nm

Intel x86_64 Microarchitectures		
µarch	Introduction	Process
Kaby Lake	2016-08-30	14 nm
Goldmont	2016-08-30	14 nm
Kittson	2017-01-01	22 nm
Skylake (server)	2017-05-04	14 nm
Coffee Lake	2017-10-05	14 nm
Goldmont Plus	2017-12-11	14 nm
Knights Mill	2017-12-18	14 nm
Palm Cove	2018-01-01	10 nm
Whiskey Lake	2018-04-01	
Amber Lake	2018-04-01	14 nm
Cannon Lake	2018-05-15	10 nm
Lakefield	2019-01-01	22 nm, 10 nm
Cascade Lake	2019-01-01	14 nm
Tremont	2019-01-01	10 nm
Snow Ridge	2019-01-01	10 nm
Sunny Cove	2019-01-01	10 nm
Ice Lake (client)	2019-05-27	10 nm
Willow Cove	2020-01-01	10 nm

Source: https://en.wikichip.org/wiki/intel/microarchitectures

What is ISA?

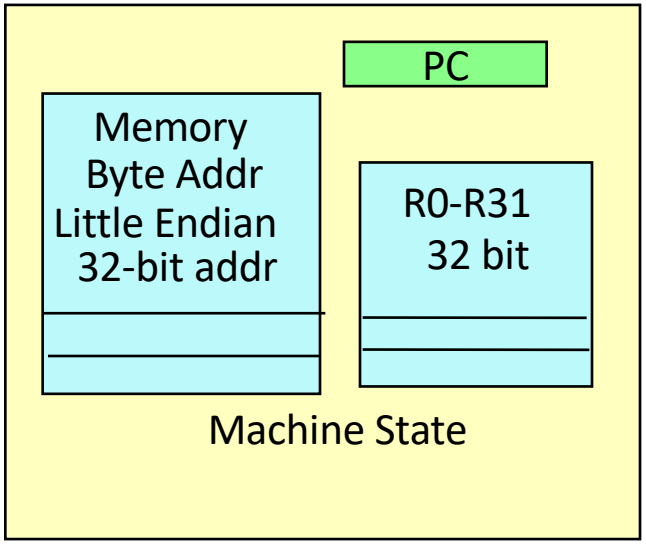
- **Architecture vs. Microarchitecture: ARM**



**** Source: arm.com***

What is ISA?

- **Machine State = Registers and Memory**
- **Registers**
  - Size/Type
    - Program Counter (= IP)
    - accumulators
    - index registers
    - general registers
    - control registers
- **Memory**
  - Visible hierarchy (if any)
  - Addressibility
    - byte, word, bit
    - byte order (endian-ness)
    - maximum size
  - protection/relocation

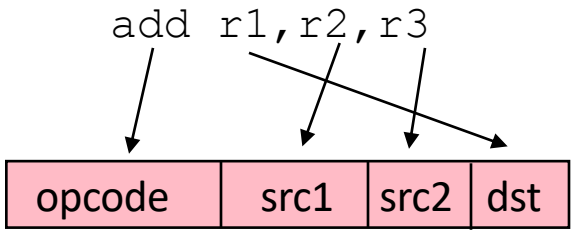


What is ISA?

- **Instruction Types**
  - ALU Operations
    - arithmetic (`add, sub, mult, div`)
    - logical (`and, or, xor, srl, sra`)
    - data type conversions (`cvtf2d, cvtf2i`)
  - Data Movement
    - memory reference (`lb, lw, sb, sw`)
    - register to register (`movi2fp, movf`)
  - 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 (`trap`)

What is ISA?

- **Components of Instructions**
  - Operations (opcodes)
  - Number of operands
  - Operand specifiers
  - Instruction encodings
  - Instruction classes
    - ALU ops (add, sub, shift)
    - Branch (beq, bne, etc.)
    - Memory (ld/st)



What is ISA?

- **Number of Operands**
  - No Operands
    ```
    HALT
    NOP
    ```
  - 1 operand
    ```
    NOT R4          R4 ⇐ ~R4
    JMP _L1
    ```
    Bitwise not.
  - 2 operands
    ```
    ADD R1, R2      R1 ⇐ R1 + R2
    LDI R3, #1234
    ```
  - 3 operands
    ```
    ADD R1, R2, R3  R1 ⇐ R2 + R3
    ```
  - 3+ operands
    ```
    MADD R4,R1,R2,R3   R4 ⇐ R1 +(R2*R3)
    ```

What is ISA?

- **Effect of Operand Number**

```
E = (C+D)*(C-D)
```

Assign

```
C ⇒ r1
D ⇒ r2
E ⇒ r3
```

3 operand machine

```
add  r3,r1,r2
sub  r4,r1,r2
mult r3,r4,r3
```

Complex Instruction

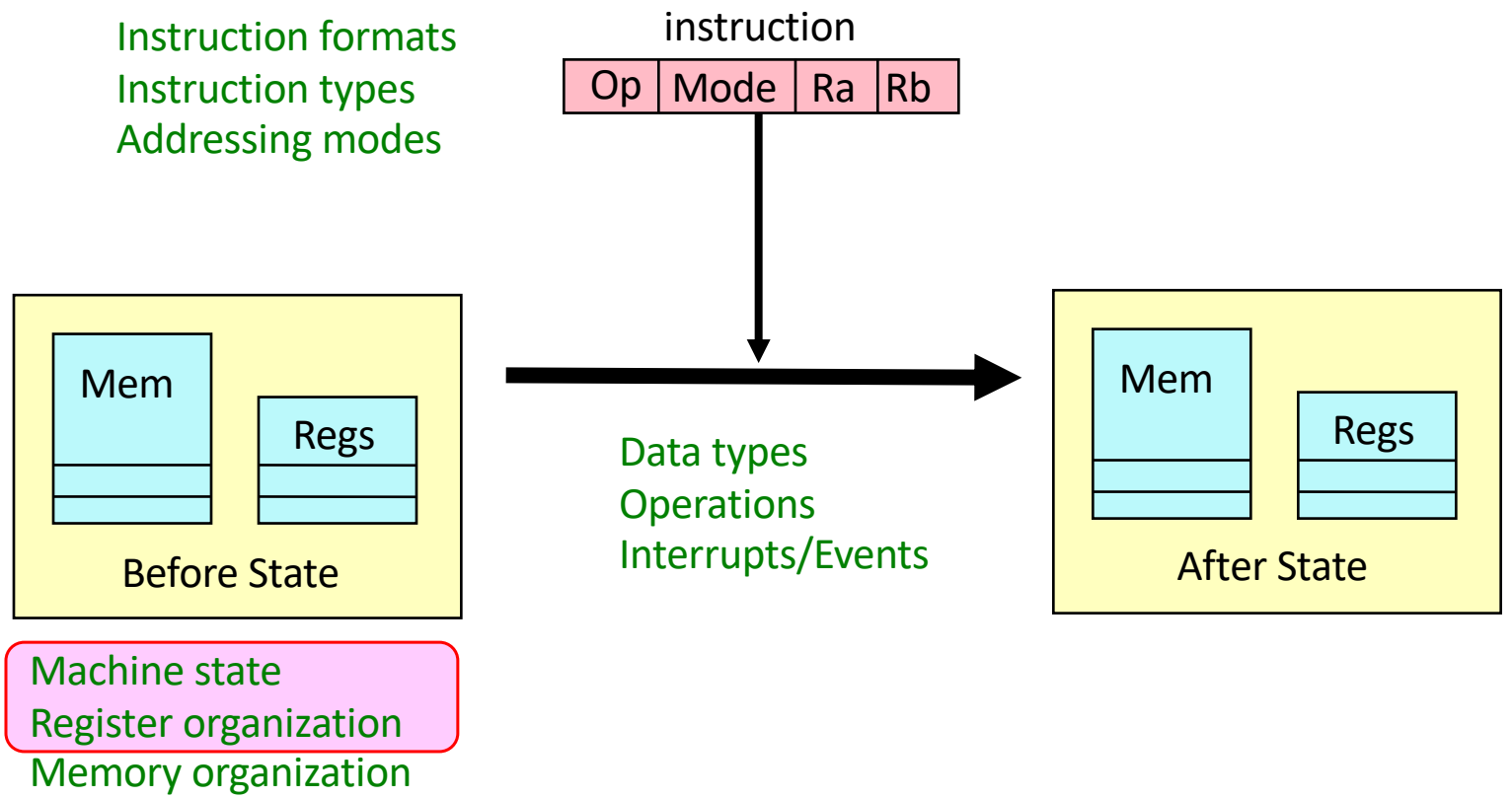
2 operand machine

```
mov  r3,r1
add  r3,r2
sub  r2,r1
mult r3,r2
```

Simple Instruction

ISA Basics
(What ISA defines?)

ISA Basics


Instruction formats
Instruction types
Addressing modes
instruction
Op
Mode
Ra
Rb
Mem
Regs
Before State
Data types
Operations
Interrupts/Events
Mem
Regs
After State
Machine state
Register organization
Memory organization

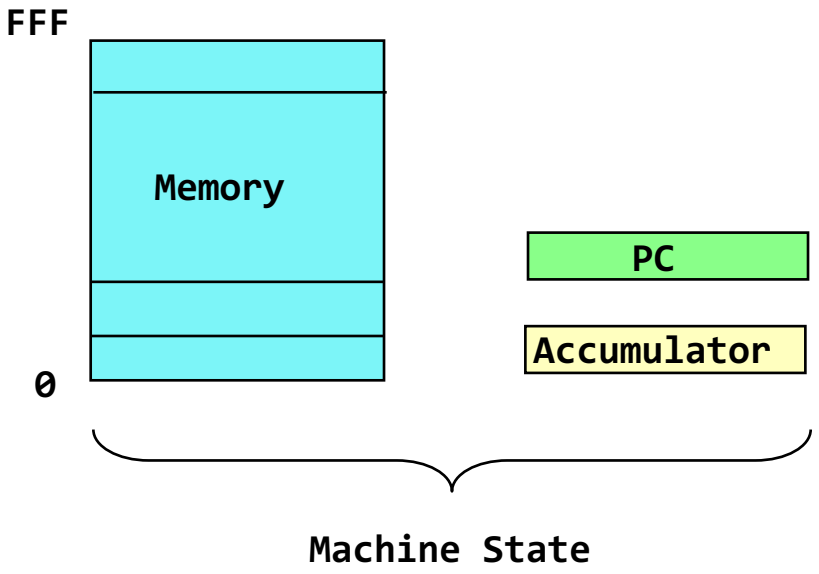
ISA Basics: Register Organization

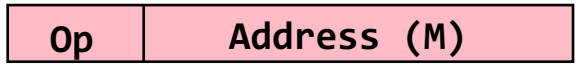
- **Evolution of Register Organization**
  - In the beginning…the accumulator
    - 2 instruction types: op and store

      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
      - *direct addressing*: *M
    - Early machines:
      - EDVAC, EDSAC…





Instruction Format

(Op encodes addressing mode)

ISA Basics: Register Organization

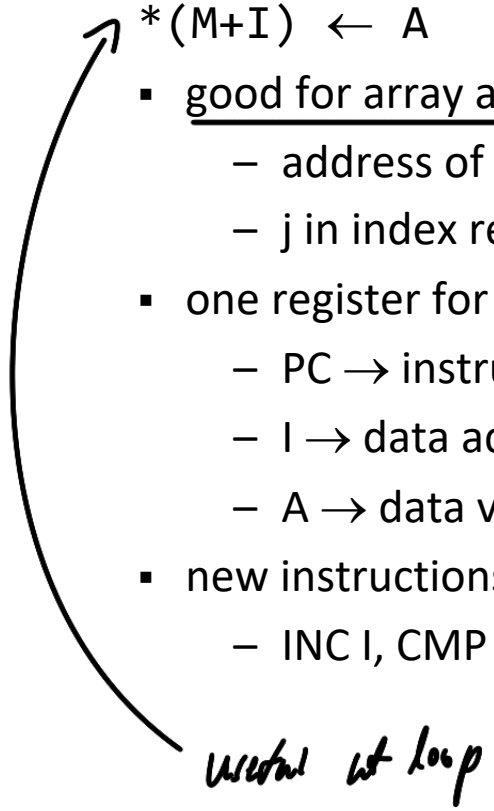
- **Why Accumulator Architectures?**
  - Registers expensive in early technologies (vacuum tubes)
  - Simple instruction decode
    - Logic also expensive
    - Critical programs were small (efficient encoding)
  - Less logic $\Rightarrow$ faster cycle time
  - Model similar to earlier calculator
  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

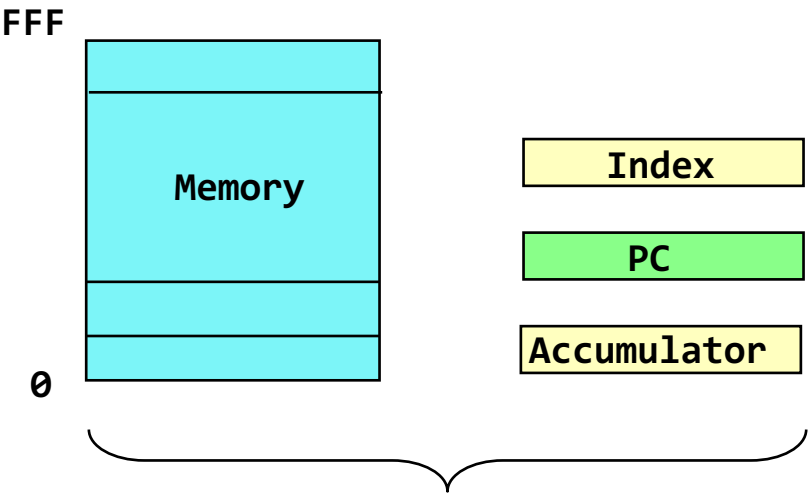
ISA Basics: Register Organization

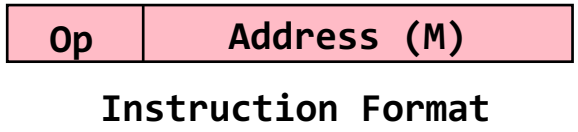
- **The Index Register**
  - Add an indexed addressing mode

    `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
      - PC → instructions
      - I → data addresses
      - A → data values
    - new instructions to use I
      - INC I, CMP I, etc.







ISA Basics: Register Organization

- **Example of Indexed Addressing**

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

```
START:  CLR     i
        CLR     sum
LOOP:   LOAD    y_addr
        AND     #MASK
        OR      i
        STORE   y_addr
        LOAD    sum
IX:     ADD     y_addr
        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

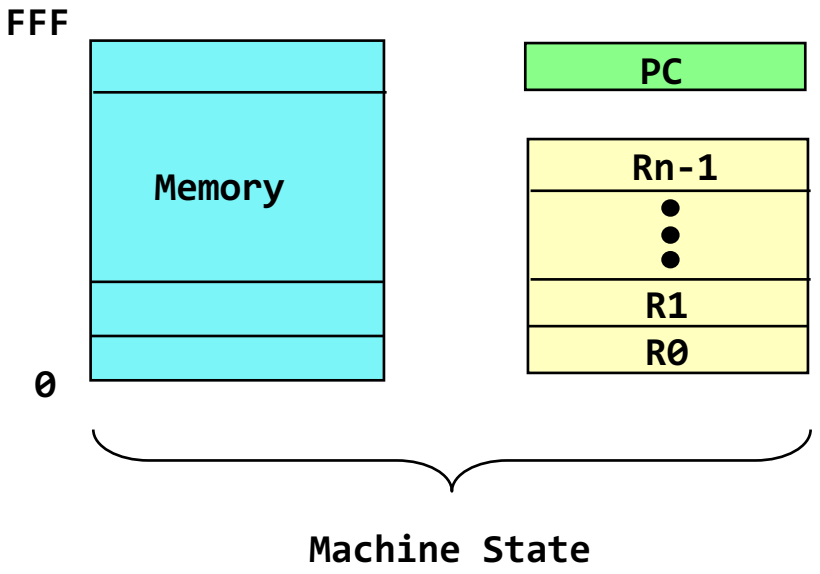
ISA Basics: Register Organization

- **But What About...**

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

ISA Basics: Register Organization

- **General Registers (GPR)!**
  - 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



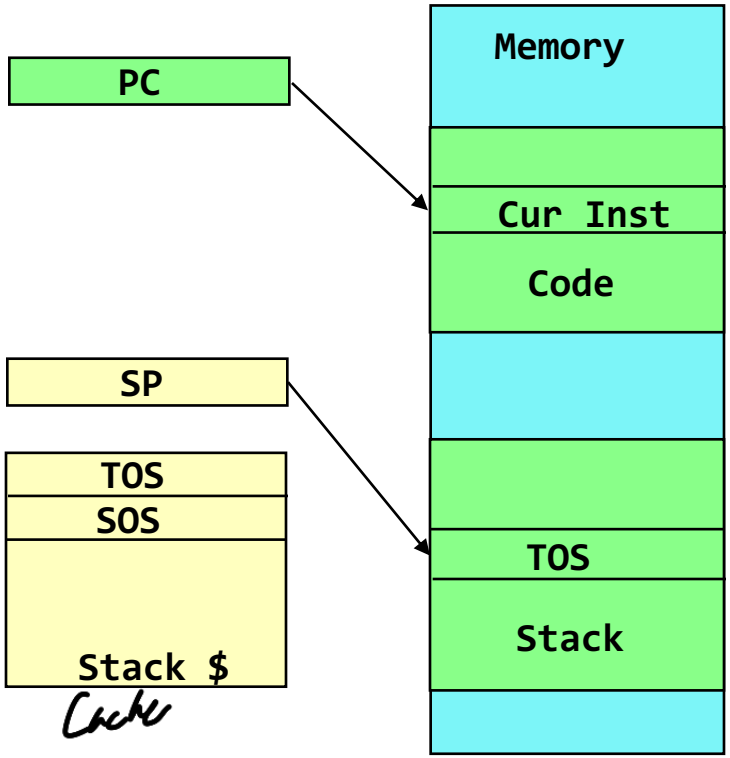
3-address Instruction Format

ISA Basics: Register Organization

- **Stack Machines**
  - Register state is PC and SP
  - All instructions performed on TOS (top of stack) and SOS (Second on Stack)

    rd is always TOS
    - pushes/pops of stack implied

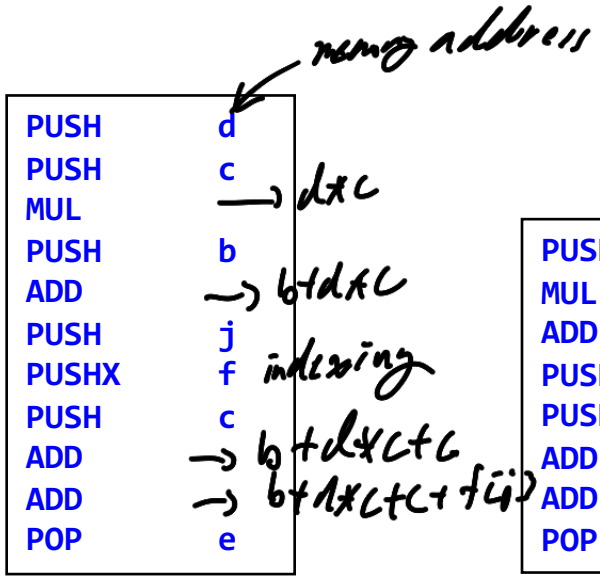
      op TOS SOS
      op TOS M
      op TOS *M
      op TOS *(M+SP)
  - Many instructions are *zero* address
  - Stack cache for performance
    - similar to register file
    - hardware managed
  - Why do we care?
    - Java Virtual Machine (JVM)



ISA Basics: Register Organization

- **Examples of Stack Code**

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



Pure Stack
(zero addresses)
11 inst, 7 addr

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

One Address Stack
8 inst, 7 addr

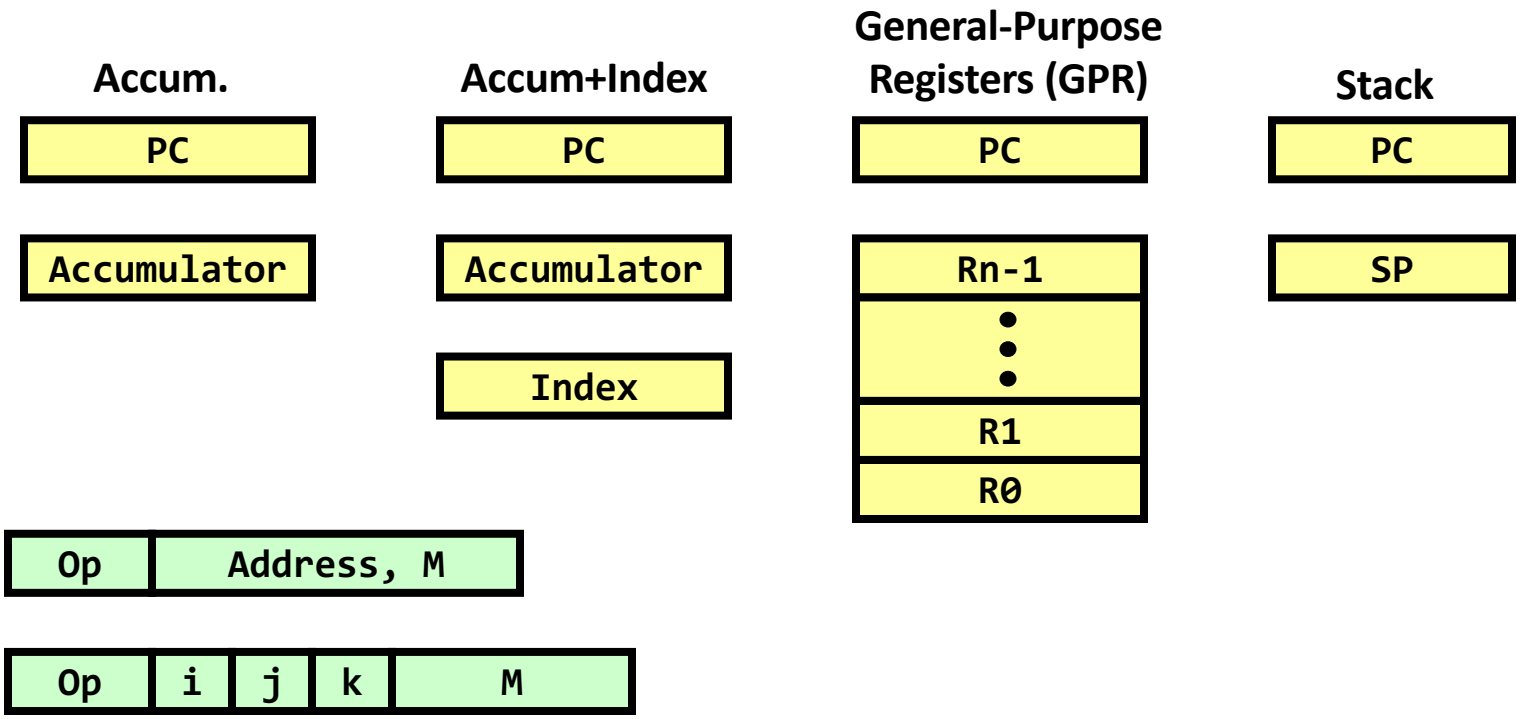
GP

```
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, 6 addr

ISA Basics: Register Organization

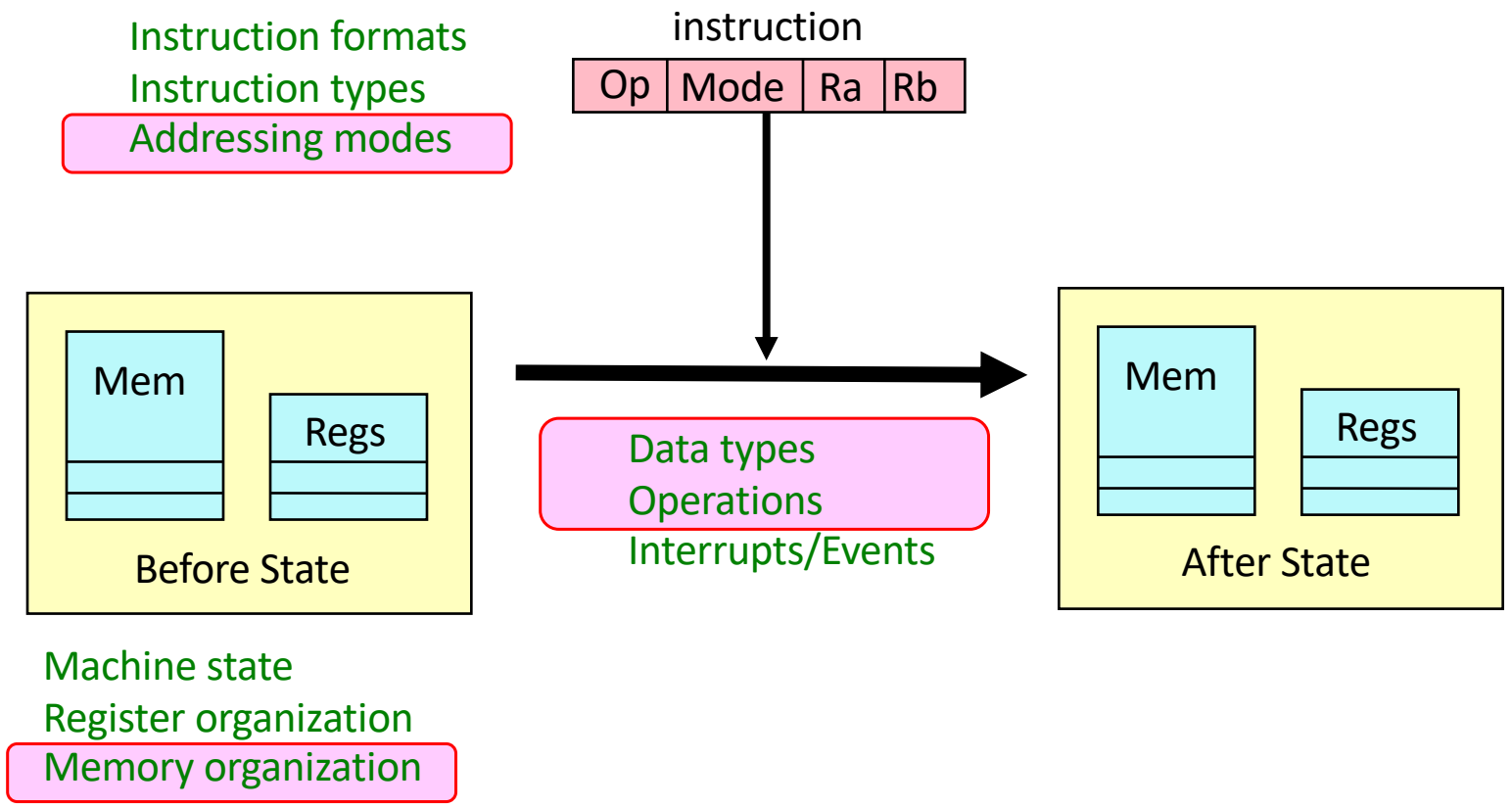
- **Review of Register Organization**



ISA Basics: Register Organization

- **Classifying ISAs: Based on how the instructions receive/produce operands**
  - Stack
    - Burroughs B5000 (1963), Java Virtual Machine (late 90s)
  - Accumulator
    - Univac-I (1951), EDSAC (1949)
  - GPR: Register-memory
    - IBM 360 (1964), DEC PDP--and later VAX (1970), Intel x86(1978)
  - GPR: Register-Register (or Load/Store)
    - IBM 801 (1974-prototype), Stanford MIPS/Berkeley RISC (mid-1980s)
    - ARM, RISC-V, IBM Power, MIPS, Sun Sparc, DEC Alpha, DSP, VLIW, etc.
  - Special case: Vector ISAs

ISA Basics

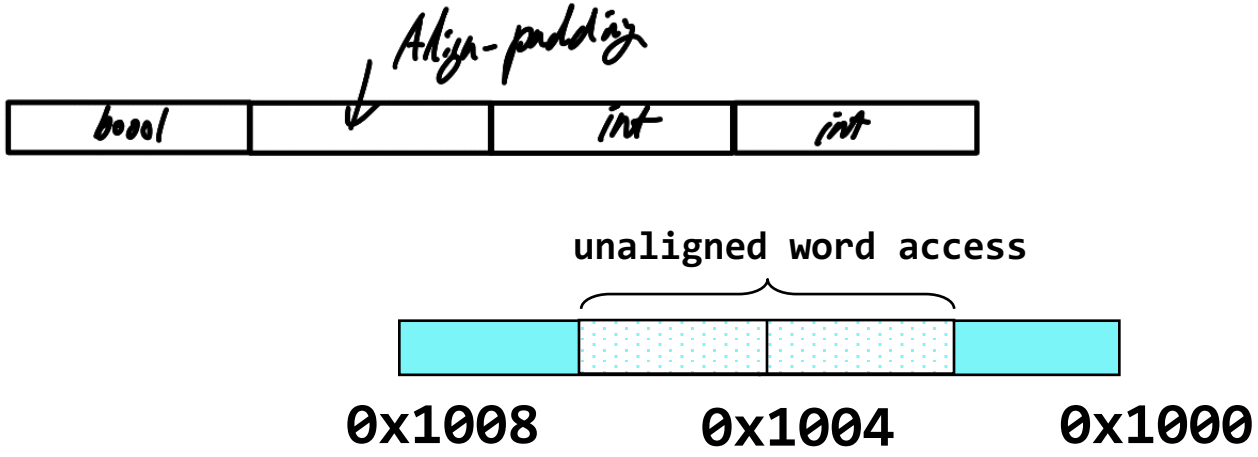

Instruction formats
Instruction types
Addressing modes
instruction
Op
Mode
Ra
Rb
Mem
Regs
Before State
Data types
Operations
Interrupts/Events
Mem
Regs
After State
Machine state
Register organization
Memory organization

ISA Basics: Memory Organization

- **Four components specified by ISA:**
  - Smallest addressable unit of memory (byte? halfword? word?)
  - Maximum addressable units of memory (doubleword?)
  - Alignment
  - Endianness

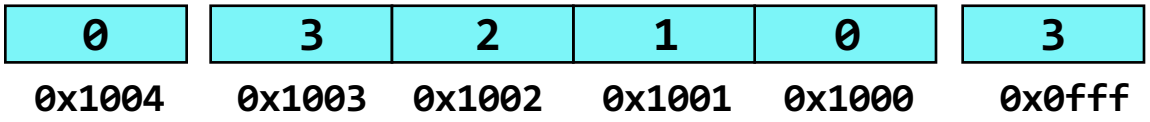
ISA Basics: Memory Organization

- **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

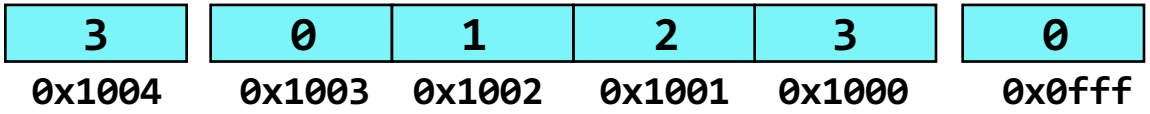


ISA Basics: Memory Organization

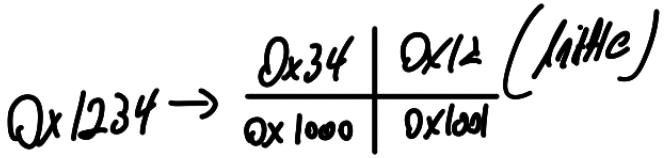
- **Endianness: How are bytes ordered within a word?**
  - Little Endian (Intel/DEC/RISC-V)



  - Big Endian (MIPS/IBM/Motorola)

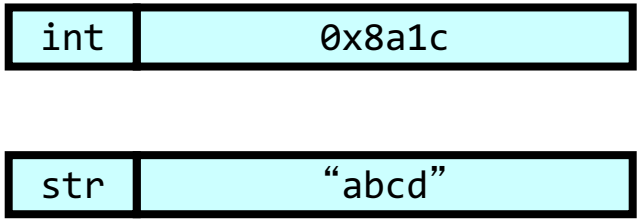


  - Today - most machines can do either (configuration register)



ISA Basics: Memory Organization

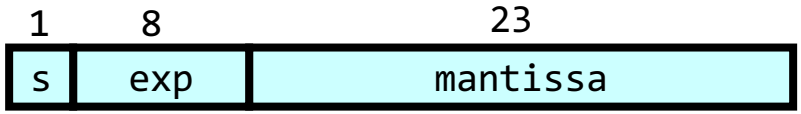
- **Data Types**
  - How the contents of memory and registers are interpreted
  - Can be identified by
    - tag
    - use
  - Driven by application
    - Signal processing
      - 16-bit fixed point (fraction)
    - Text processing
      - 8-bit characters
    - Scientific computing
      - 64-bit floating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



ISA Basics: Memory Organization

- **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

ISA Basics: Addressing Modes

- **Addressing Modes Driven by Program Usage**

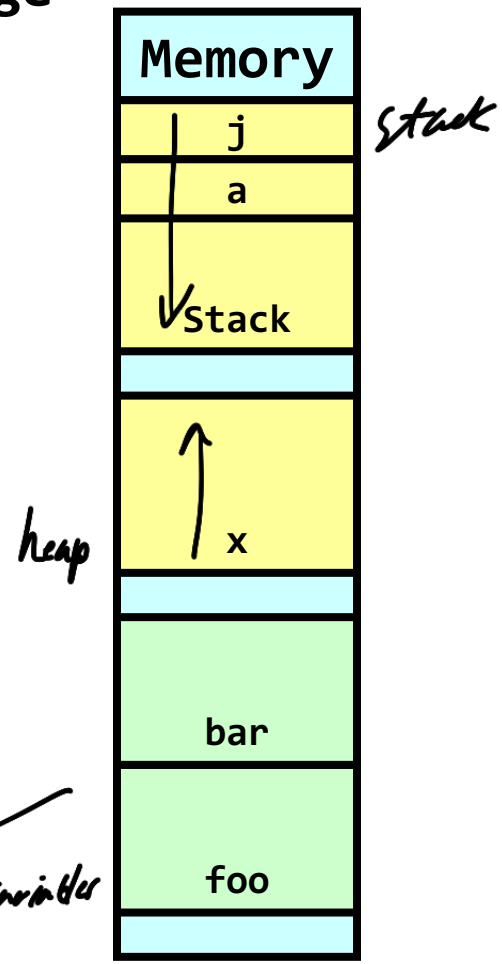
```
double x[100] ;                  // global
void foo(int a) {                // argument
  int j ;                        // local
  for(j=0;j<10;j++)
    x[j] = 3 + a*x[j-1] ;
  bar(a);
}
```

procedure

constant

argument

array reference



ISA Basics: Addressing Modes

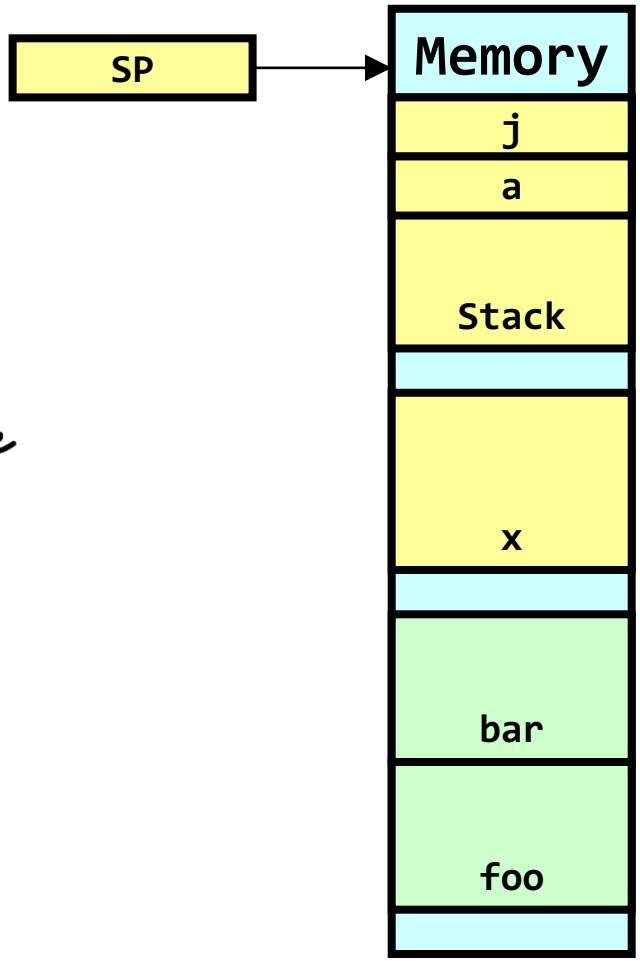
- **Addressing Modes**
  - Stack relative for locals and arguments

    a, j: *(R30+x)
  - Short immediates (small constants)

    3 – literal values
  - Long immediates (global addressing)

    &x[0], &bar: 0x3ac1e400 address value
  - Indexed for array references

    *(R4+R3)

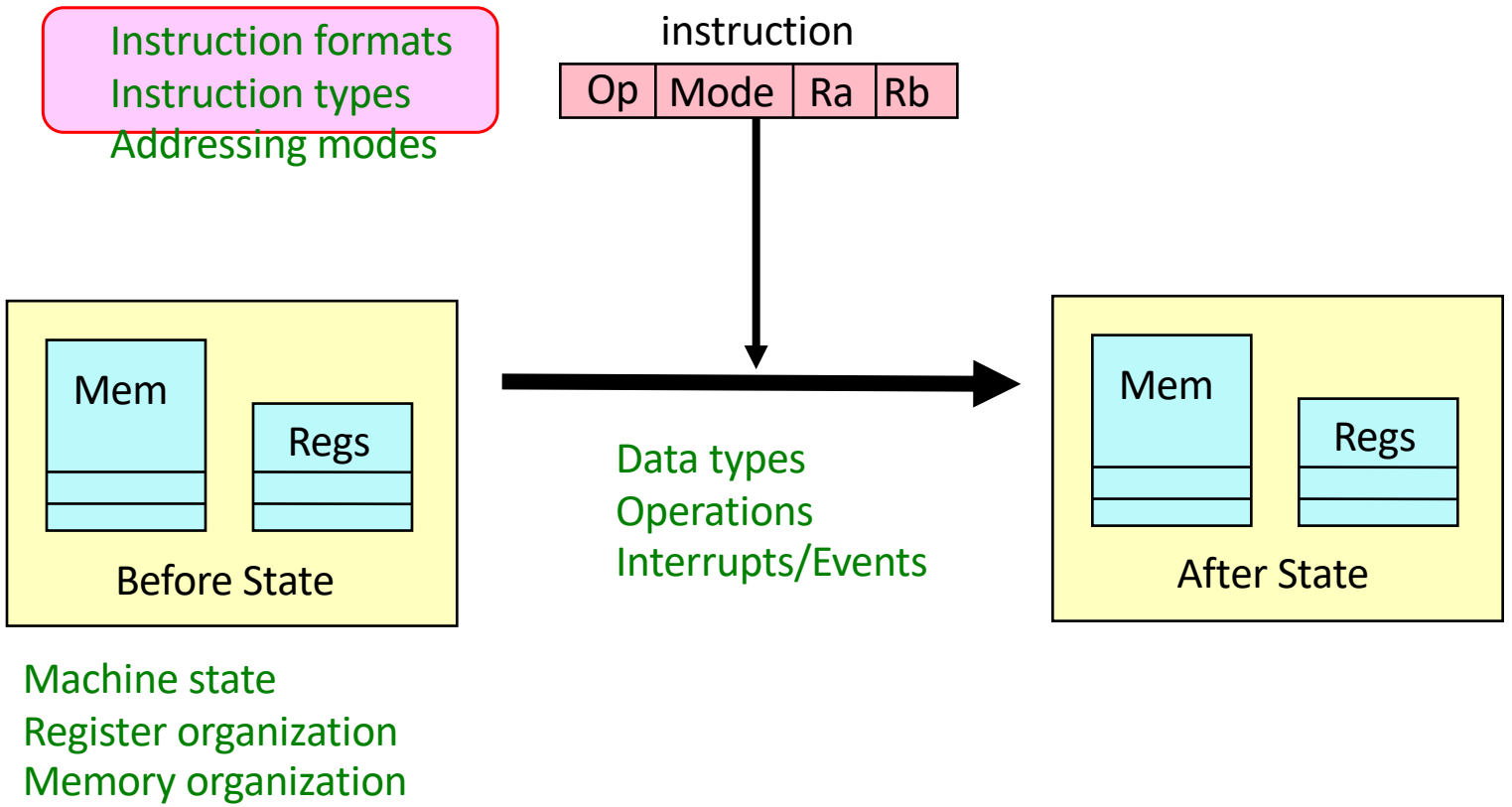


ISA Basics: Addressing Modes

- **Addressing Mode Summary with Examples**

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1,–(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

ISA Basics


Instruction formats
Instruction types
Addressing modes
instruction
Op
Mode
Ra
Rb
Mem
Regs
Before State
Data types
Operations
Interrupts/Events
Mem
Regs
After State
Machine state
Register organization
Memory organization

ISA Basics: Instruction Types

- **Control Instructions**
  - Implicit control on each instruction

    PC ← PC + 4
  - Unconditional jumps

    PC ← X (direct)

    PC ← PC + X (PC relative)

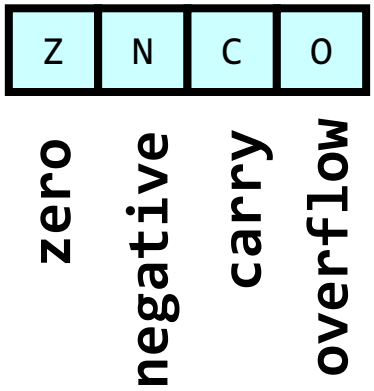
    X can be constant or register
  - Conditional jumps (branches)

    PC ← PC + ((cond) ? X : 4)
  - Predicated instructions
  - Conditions
    - flags
    - in a register
    - fused compare and branch

```
LOOP:   LOAD    R1 <- (R5+R2)
        ADD     R3 <- R3 + R1
        ADD     R2 <- R2 + 4
        CMP     R4 <- R2 == 8
        JNE     R4, LOOP
```

ISA Basics: Instruction Types

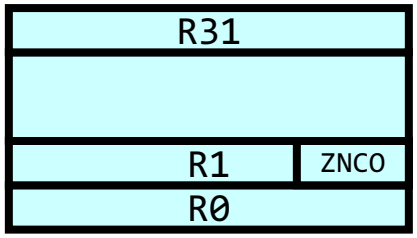
- **Representing Conditions - Flags**
  - Traditional approach
  - Record ALU status from each op
    - Z, N, C, O
  - Single copy, modified by most instructions
    - Compare and branch must be kept together
    - Single flag register can be a serial bottleneck



ISA Basics: Instruction Types

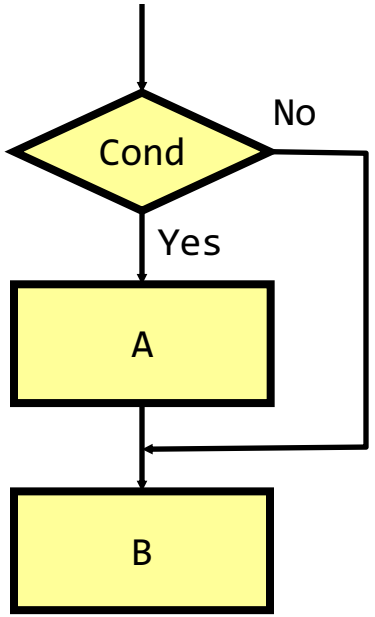
- **Storing Conditions in Register**
  - Record ALU status from compare in a register
    - `R1 ← COMP(R2, R3)`
  - Alternatively use *condition registers*
  - Test the register later for branch
    - `BGE R1, LOOP`
  - Allows separation of test and branch
  - Enables parallel execution

```
R1 ← COMP(R2, R3)
R4 ← COMP(R5, R3)
BGE R1, DST1
BGE R4, DST2
...
```



ISA Basics: Instruction Types

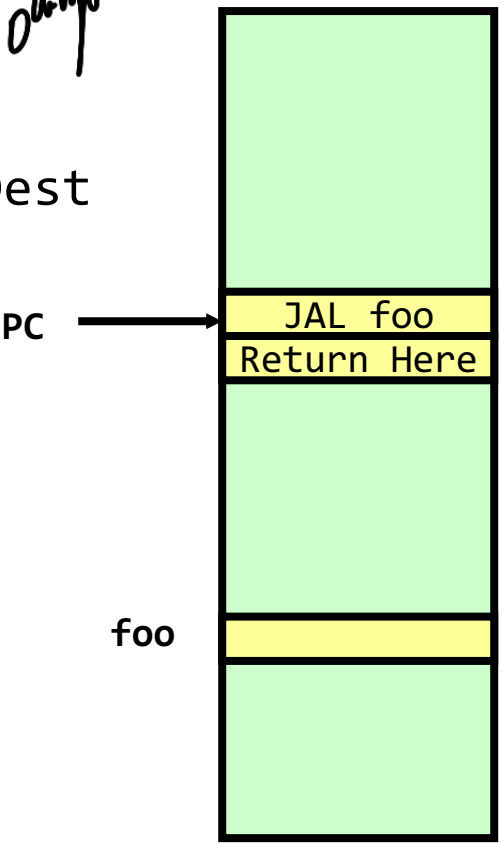
- **Conditional Execution**
  - Fused compare and branch (compt&jump)
    - BGE R1, R2, LOOP
    - reduces instruction count
    - forces compare and branch to be together
    - complicates pipelining
  - Predication
    - disable an instruction based on a condition
      - (if GE C0) ADD R1, R2, R3



ISA Basics: Instruction Types

- **Support for Procedures**
  - Branch and Link
    - store return address in reg and jump

      `JALR Rdest:  Rx ← PC + 4, PC ← Dest`
  - Subroutine call
    - push return address on stack and jump
  - CALLP (VAX)
    - push return address
    - set up stack frame
    - save registers
    - ...

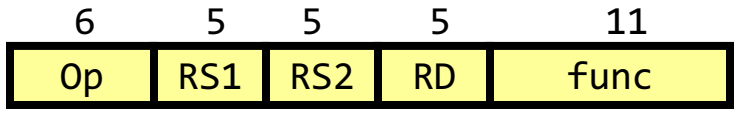


ISA Basics: Instruction Formats

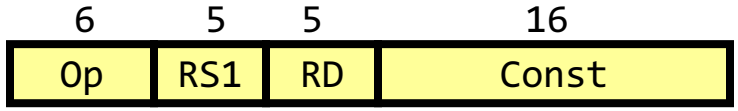
Instruction Formats

- Different instructions need to specify different information
  - return
  - increment R1
  - R3 ← R1 + R2
  - jump to 64-bit address
- Frequency varies
  - instructions
  - constants
  - registers
- Can encode
  - fixed format
  - small number of formats
  - byte/bit variable

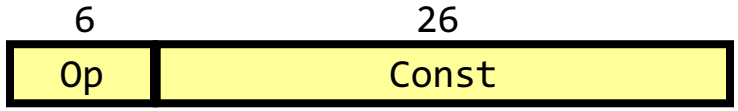
R: rd ← rs1 op rs2



I: ld/st, rd ← rs1 op imm, branch



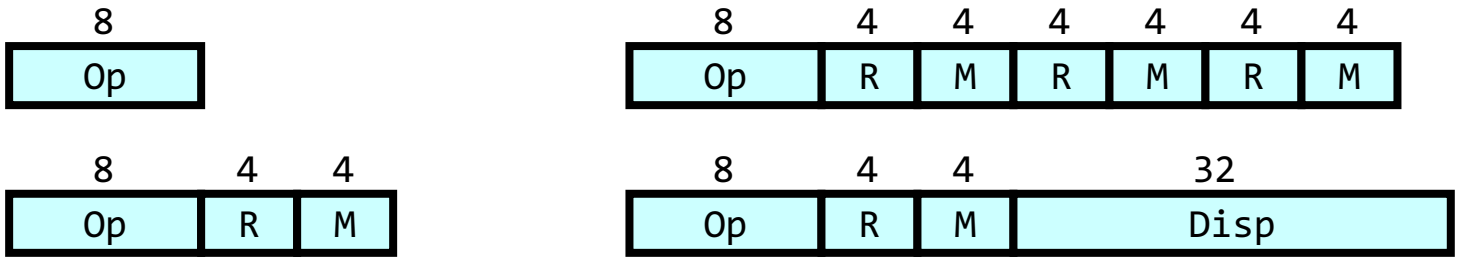
J: j, jal



Fixed-Format (MIPS)

ISA Basics: Instruction Formats

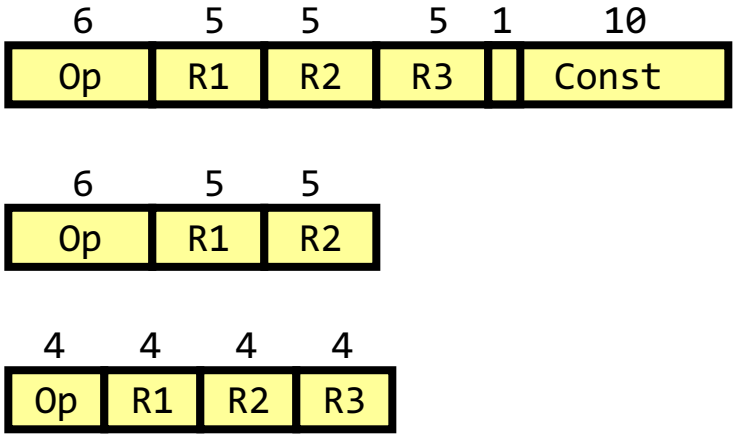
- **Variable-length instructions: give more efficient encodings**
  - no bits to represent unused fields/operands
  - can frequency code operations, operands, and addressing modes
  - Examples
    - VAX-11, Intel x86 (byte variable)
    - Intel 432 (bit variable)
- **But - can make fast implementation difficult**
  - sequential determination of location of each operand



VAX instrs: 1-53 bytes!

ISA Basics: Instruction Formats

- **Compromise: A Few Good Formats**
  - Gives much better code density than fixed-format
    - important for embedded processors
  - Simple to decode



ISA Principles: CISC vs RISC

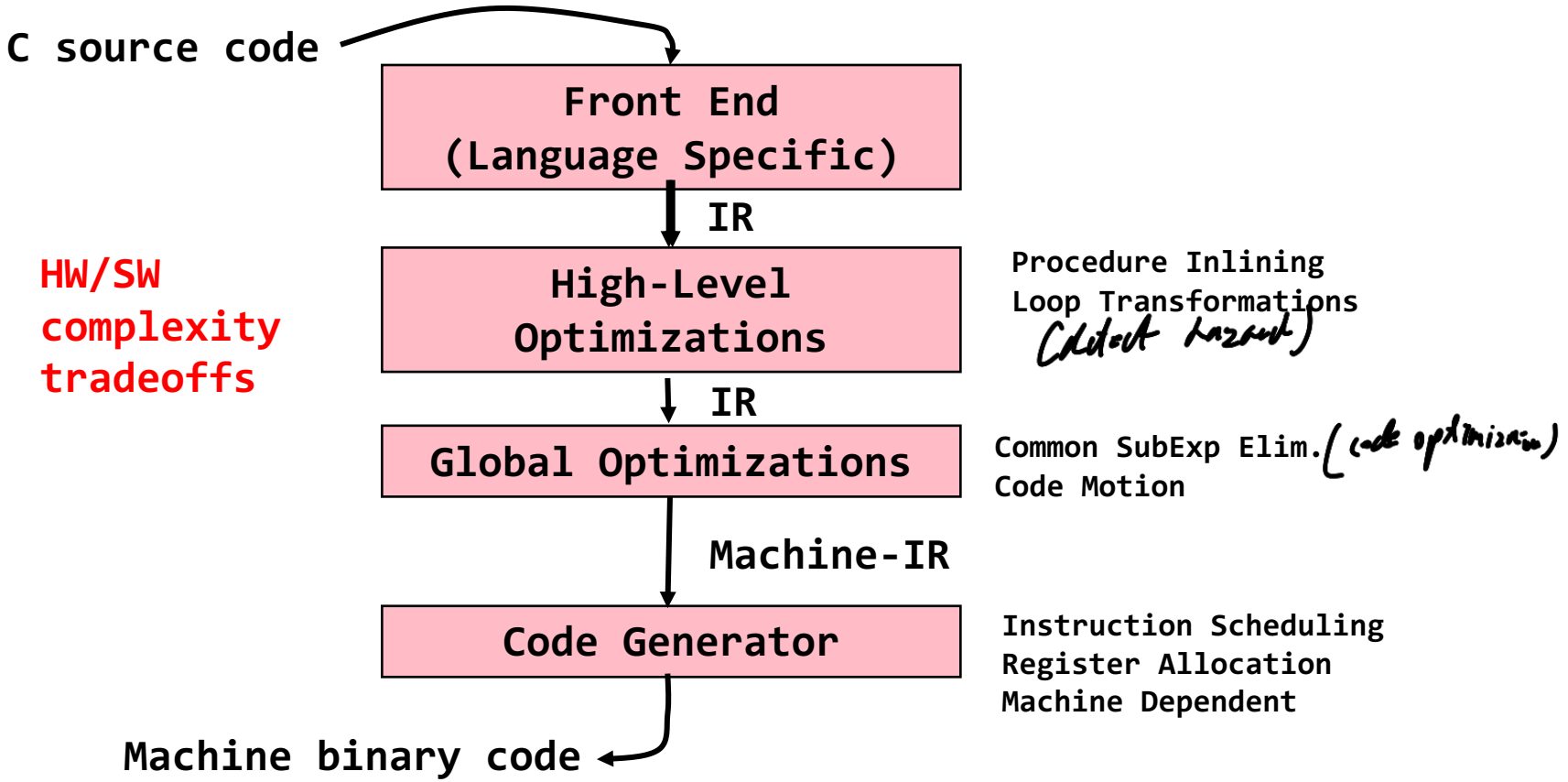
- **What is a RISC?**
  (Reduced Instruction Set Computer)
  - no firm definition
  - generally includes
    - general registers
    - fixed 3-address instruction format
    - strict load-store architecture
    - simple addressing modes
    - simple instructions
  - Examples
    - RISC-V
    - DEC Alpha
    - MIPS
  - Advantages
    - good compiler target
    - easy to implement/pipeline

internal μArch change to RISC

- **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

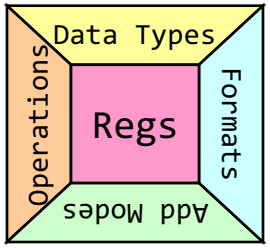
ISA Principles

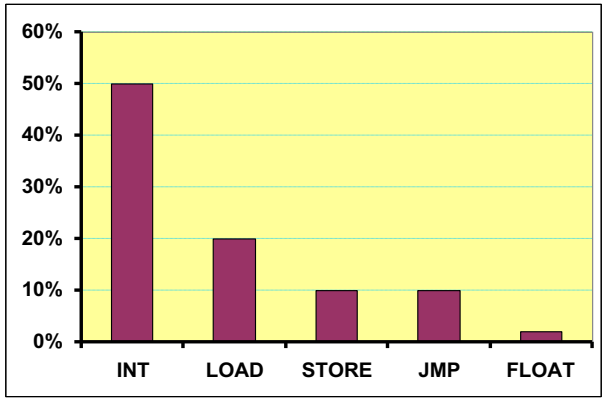
- Role of the Optimizing Compiler



ISA Principles : Virtue of engineering — simple / modular / common

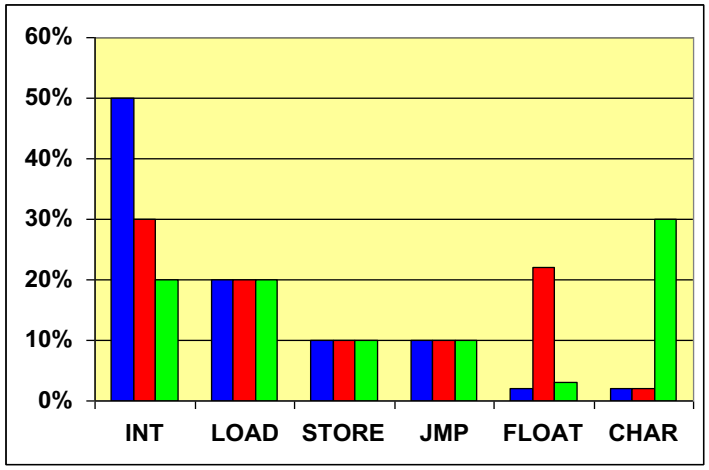
- **Principles of Instruction Set Design (1)**
  - Keep it simple, stupid! (KISS)
    - complexity
      - increases logic area
      - increases pipe stages
      - increases development time
    - evolution tends to make kludges
  - Orthogonality (modularity)
    - simple rules, few exceptions
    - all ops on all registers
  - Frequency
    - make the common case fast
      - some instructions (cases) are more important than others

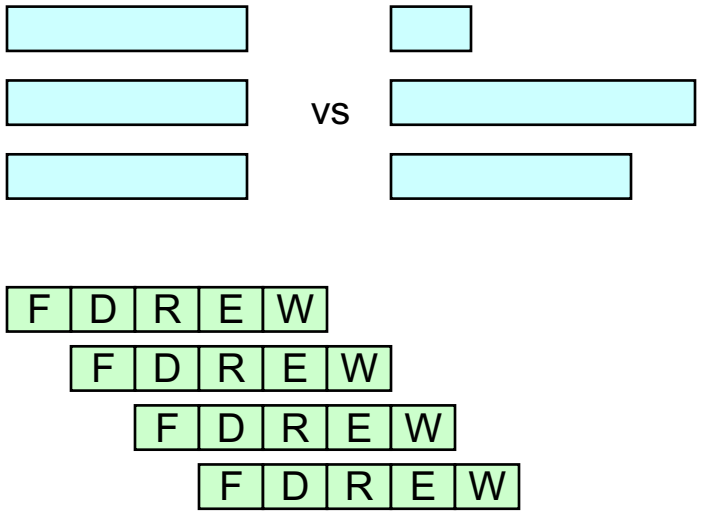




ISA Principles

- **Principles of Instruction Set Design (2)**
  - Generality
    - not all problems need the same features/instructions
    - principle of *least surprise*
    - performance should be easy to predict
  - Locality and concurrency
    - design ISA to permit efficient implementation
      - today
      - 10 years from now





ISA Summary

- **Good ISA design**
  - KISS! - only implement necessities (encodings, address modes, etc.)
  - FOG: Frequency, Orthogonality, Generality
- **Instruction Types**
  - ALU ops, Data movement, Control
- **Addressing modes**
  - Matched to program usage (local vars, globals, arrays)
- **Program Control**
  - Conditional/unconditional branches and jumps
  - Where to store conditions
  - PC relative and absolute