# Computer Architecture

## Instruction Set Architecture

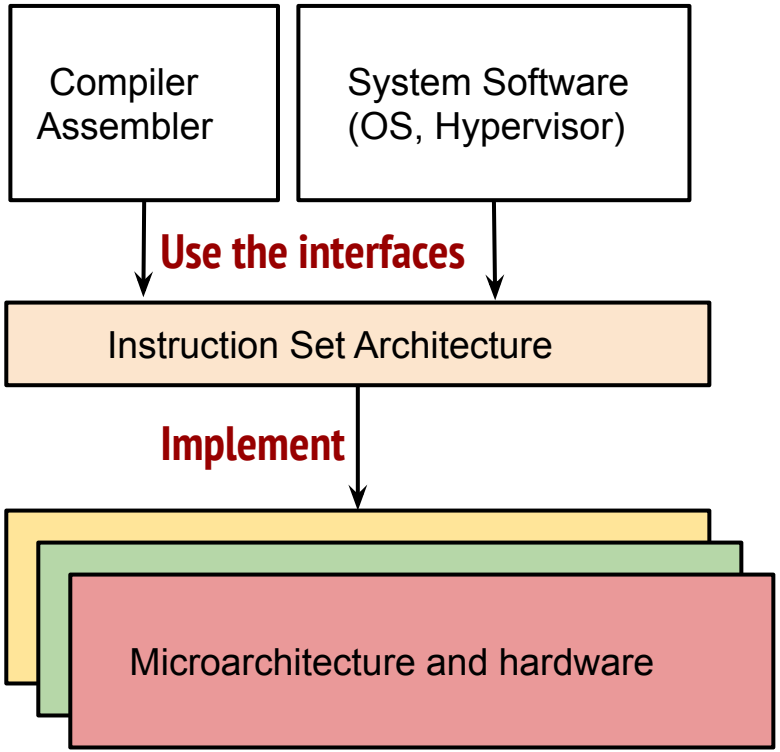
Debadatta Mishra, CSE, IITK

# Simulation using Champsim (Recap)

- Champsim is a trace-based simulator
- Features: x86 traces, OoO CPU, cache/memory
- Gives a specialized pintool for tracing
- Example:
  - Change L1 DCache size for a simple benchmark
  - L1 DCache hit improves with increased cache size, but no improvement in IPC!
  - Changing OoO parameters makes the effects visible…

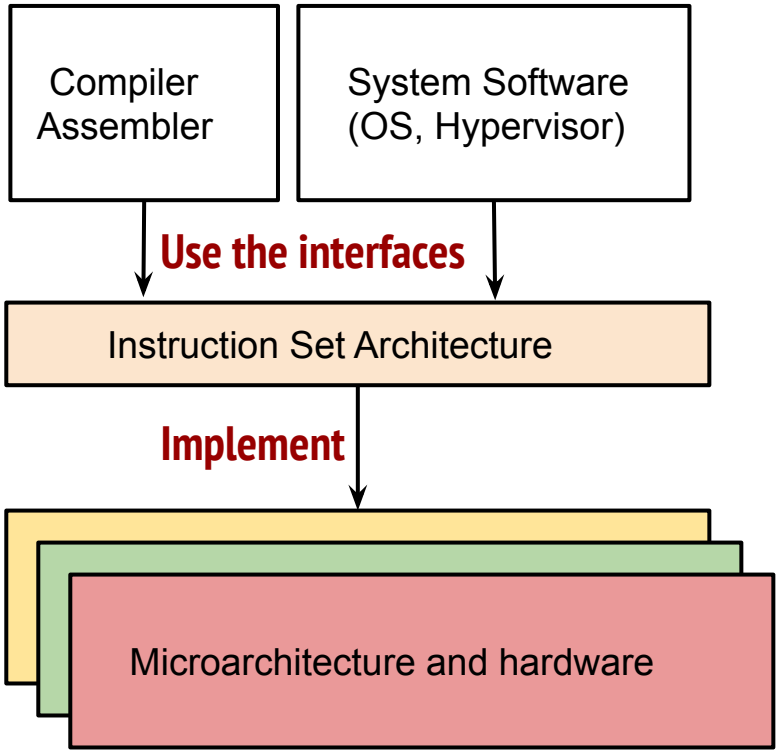

???

# Instruction Set Architecture




- ISA is an abstraction with clearly defined interface between the hardware and software
- Interface includes
  - Software visible state and semantics
  - Operations and their impacts on the state

# Instruction Set Architecture

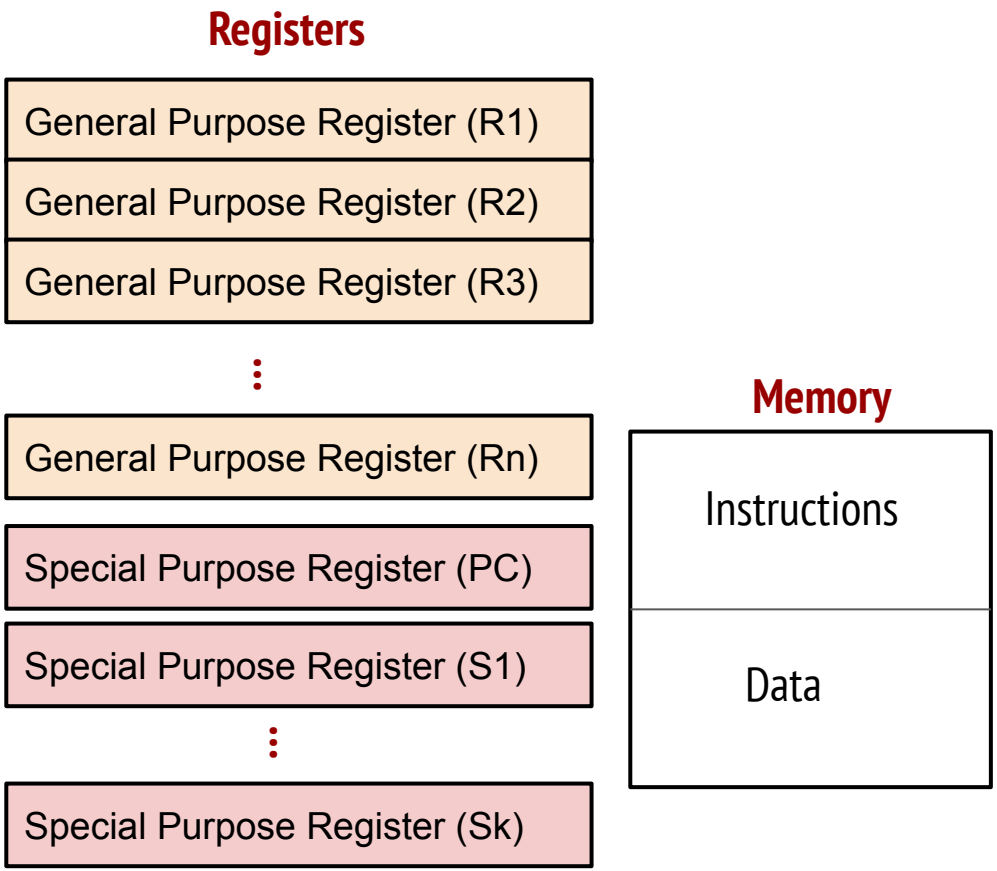



- ISA is an abstraction with clearly defined interface between the hardware and software
- Interface includes
  - Software visible state and semantics
  - Operations and their impacts on the state

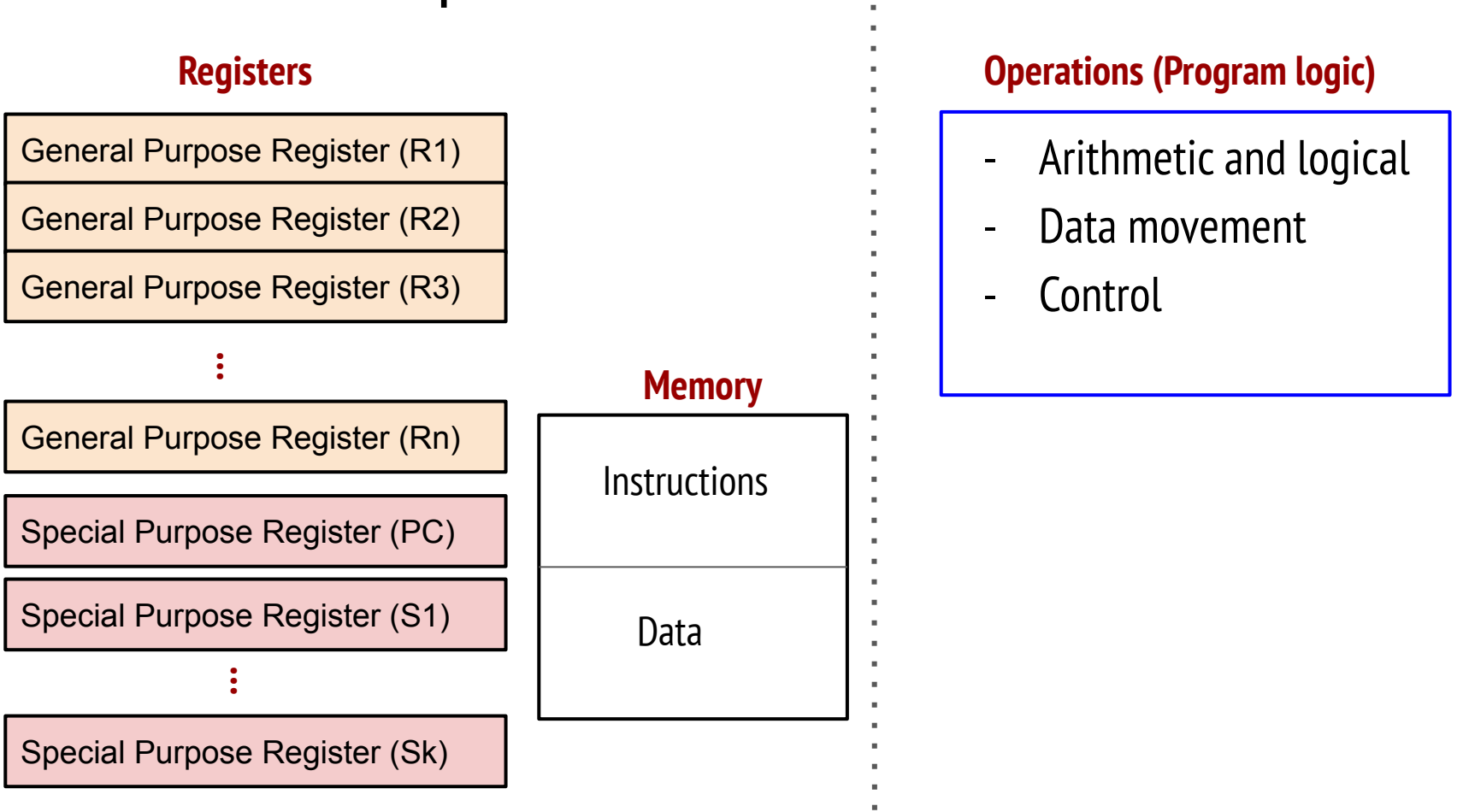
May not expose states introduced for implementation

ISA does not specify how operations are implemented!

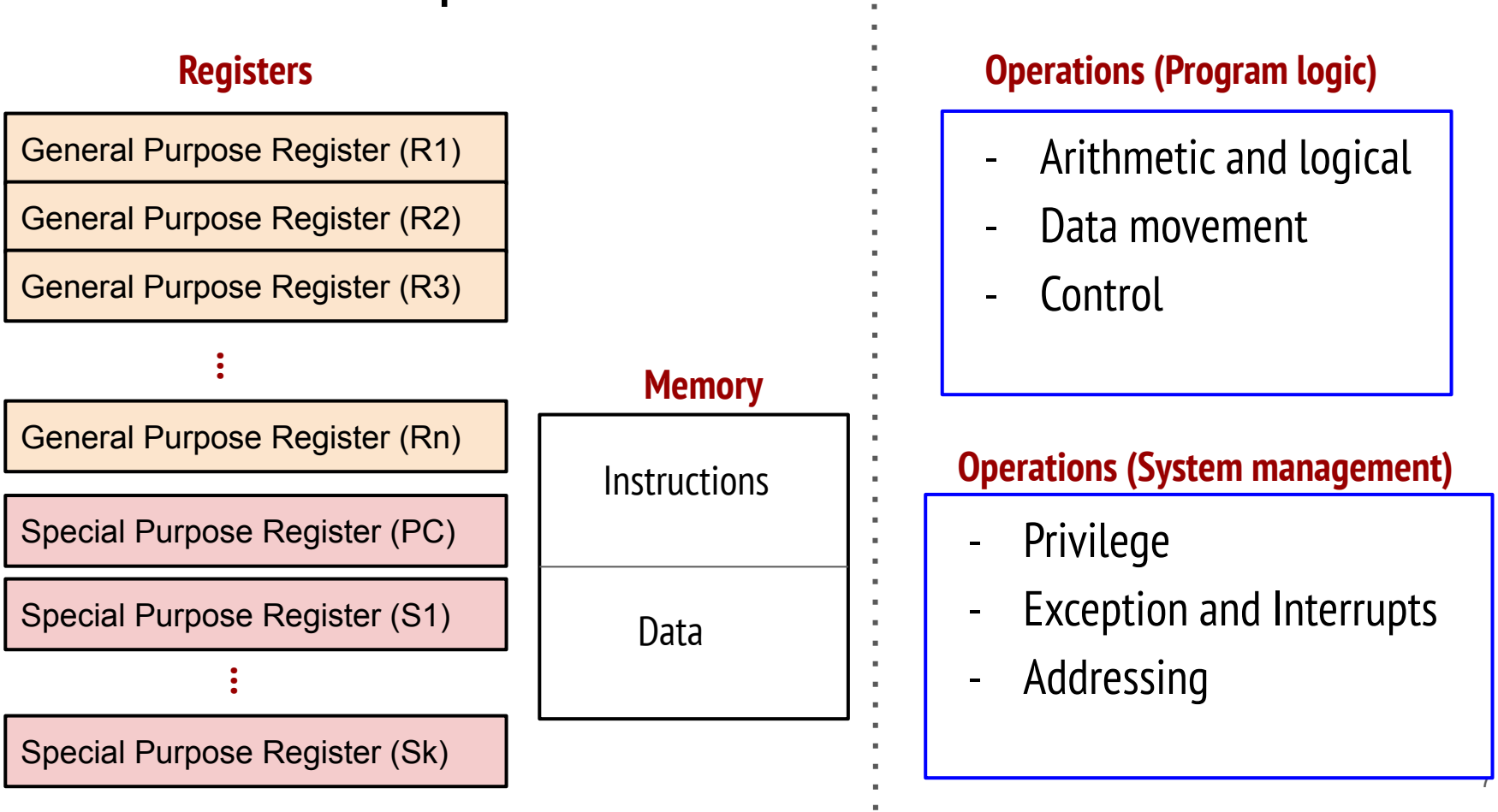
# ISA: State and Operations



# ISA: State and Operations

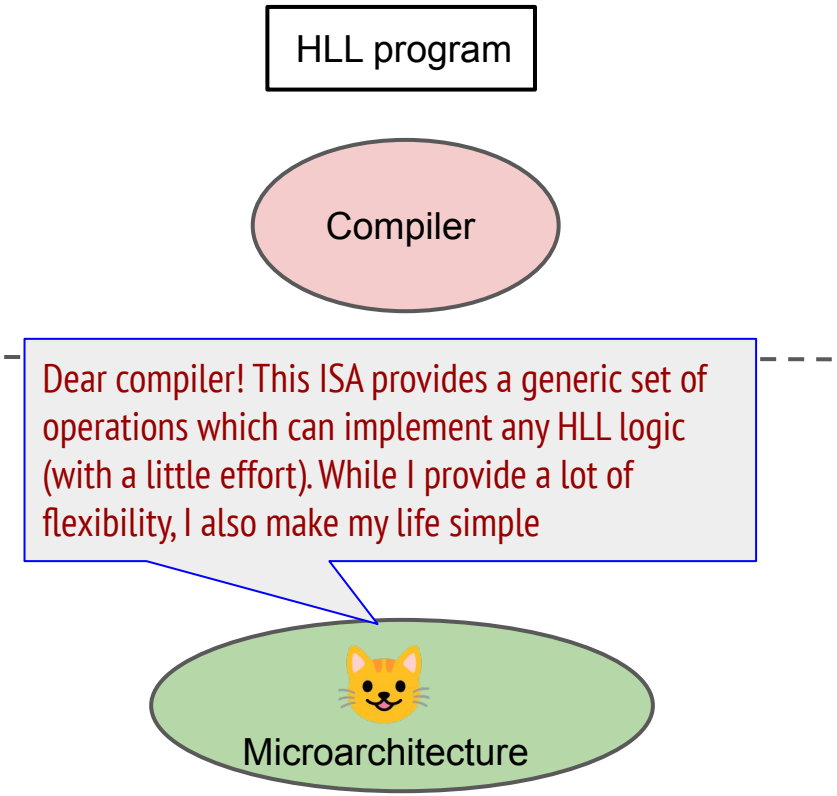


# ISA: State and Operations



# ISA design consideration: Division of responsibility


**Approach 1: Rich and Specialized ISA**




**Approach 2: Simple and generic ISA**

# Illustration: Initialize a memory variable

**Approach 1: Rich and Specialized ISA**

- Initialize a memory location to a particular value
  - ISA support: Move immediate to memory
- Example (X86): `mov` instruction with immediate operand

**Approach 2: Simple and Generic ISA**

- Initialize memory location to a particular value
- ISA support
  - Place immediate in a register using arithmetic operation and store it to the specified memory address
- Example (MIPS ISA): `add,r0,sw`

Which approach you like and why?

# Illustration: Memory to memory copy (memcpy)

**Approach 1: Rich and Specialized ISA**

- Special instruction to copy **N** bytes from memory addresses **S** to memory address **D**
- ISA specification
  - Place **S**, **D** and **N** in *specially designated registers*
- Example (x86 ISA): Special instruction `mov(s/b/w)` with a prefix `rep`

**Approach 2: Simple and Generic ISA**

- Compile **memcpy** to an assembly loop
- ISA support
  - Memory Load/Store
  - Conditional branch
- Example (MIPS ISA): Use instruction like `lw, sw, bne`

Which approach you like and why?

# Illustration: Function invocation

**Approach 1: Rich and Specialized ISA**

- Hardware support for callee saved procedure invocation (+ Simple ISA)
- ISA specification: Specify the registers to be saved as a mask in the procedure and the # of arguments
- Example (VAX ISA): Special instruction **`calls`** and **`ret`**

**Approach 2: Simple and Generic ISA**

- A designated register for SP and hardware support for saving the return address onto the stack
- ISA specification: Arguments passed in register (+stack), no mandated calling convention, return address in a register
- Example (MIPS ISA): Use instruction like **`jal`** and **`jr`**

Which approach you like and why?

# Illustration: Addressing (flat vs. segmented)

**Approach 1: Rich and Specialized ISA**

- Provide special segment registers as base for different segments of a program (code, head, stack etc.)
- ISA specification
  - Load segment registers with proper base address and use displacement addressing
- Example (x86 ISA): **mov DS:0x100, EAX** after loading the base address to **DS**

**Approach 2: Simple and Generic ISA**

- ISA does not provide any notion of segments, the software may choose to implement using GPRs
- ISA support
  - Memory Load/Store
  - Displacement addressing
- Example: The software can use any register as the base and load/store

Which approach you like and why?

# Illustration: Context switch between privilege levels

**Approach 1: Rich and Specialized ISA**

- Special memory block to save and restore the state of the user context
- ISA specification
  - The memory block address must be specified before the process is scheduled in user mode
- Example (x86 ISA): Special context switch support using kernel stack page

**Approach 2: Simple and Generic ISA**

- Minimal #of special registers
- ISA support
  - Stores the value of the user PC and cause of exception into special registers
- Example (MIPS ISA): `Cause` and `ra` registers

Which approach you like and why?

# ISA Implications: Instruction encoding

- Number of instructions
  - More instructions (opcodes) are required in specialized ISA but *less # of instructions* are required to convert HLL to assembly
  - Less instructions ⇒ Better performance ?

# ISA Implications: Instruction encoding

- Number of instructions
  - More instructions (opcodes) are required in specialized ISA but *less # of instructions* are required to convert HLL to assembly $ExecutionTime = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Seconds}{Cycle}$
  - Less instructions ⇒ Better performance ? Not necessarily true
- Information required to specify instructions

# ISA Implications: Instruction encoding

- Number of instructions
  - More instructions (opcodes) are required in specialized ISA but *less # of instructions* are required to convert HLL to assembly $ExecutionTime = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Seconds}{Cycle}$
  - Less instructions ⇒ Better performance ? Not necessarily true
- Information required to specify instructions
  - More (and varying amount of) information is required in complex ISAs
  - Complex encoding ⇒ Complex decoding

# ISA Implications: Instruction encoding

- Number of instructions
  - More instructions (opcodes) are required in specialized ISA but *less # of instructions* are required to convert HLL to assembly
  - Less instructions ⇒ Better performance ? Not necessarily true
- Information required to specify instructions
  - More (and varying amount of) information is required in complex ISAs
  - Complex encoding ⇒ Complex decoding
- Registers
  - Complex ISAs have more specialized registers
  - Implications?

# ISA Implications: Instruction encoding

- Number of instructions
  - More instructions (opcodes) are required in specialized ISA but *less # of instructions* are required to convert HLL to assembly
  - Less instructions ⇒ Better performance ? Not necessarily true
- Information required to specify instructions
  - More (and varying amount of) information is required in complex ISAs
  - Complex encoding ⇒ Complex decoding
- Registers
  - Complex ISAs have more specialized registers
  - Implications: Compiler's job becomes difficult, increased size of register file

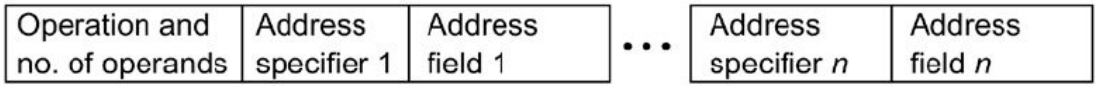
# Implications: Hardware complexity

- Decoding
  - Complex decoding logic for variable-length instructions
- Control
  - Complex control path for complex ISAs
  - Hardwired control vs. Micro-programmed control
- Single cycle architecture
  - Cycle time determined by?

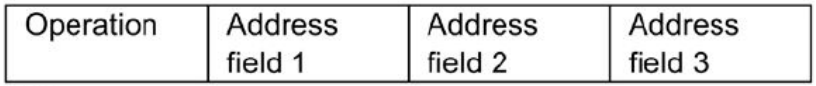
# Implications: Hardware complexity

- Decoding
  - Complex decoding logic for variable-length instructions
- Control
  - Complex control path for complex ISAs
  - Hardwired control vs. Micro-programmed control
- Single cycle architecture
  - Cycle time determined by? Longest execution cycle
- Pipelined architecture
  - Deep pipelines (more stages) to handle complex instructions
  - New ideas to tackle complexity ⇒ OoO data path

# Instruction encoding [1]

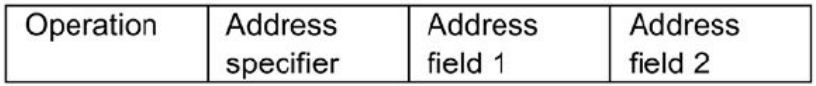



(A) Variable (e.g., Intel 80x86, VAX)




(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)






(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

- Fixed: operations and addressing modes are part of opcode
- Variable: #of operands, type and addressing mode of operands are variable in complex ISA ⇒ explicit specification required
- Hybrid: instruction length is part of opcode

1. Figure A.18 H & P

# Instruction encoding: MIPS vs. X86

MIPS

```
R-type:  op|rs|rt|rd|shamt| funct
         6 |5 | 5|5 | 5    |  6

Example: add $s0, $s1, $s2
Encoding: 0|17|18|16|0|32

I-type:  op |rs|rt|imm
          6 | 5| 5 |16

J-type: op | addr

         6 |  26
```

X86_64

```
7e4:48 c7 c0 20 00 00 00   mov  $0x20,%rax

7eb:48 c7 c1 20 10 00 00   mov  $0x1020,%rcx

7f2:48 89 c8               mov    %rcx,%rax

7f5:48 31 c0               xor    %rax,%rax

7f8:48 8b 4d f8          mov  -0x8(%rbp),%rcx

7fc:48 01 c1             add    %rax,%rcx
```

# RISC vs. CISC

- RISC: Flaw is fundamental ⇒ CISC: No can be solved using more transistors
- RISC: Hardware design becomes very complex ⇒ CISC: Compatibility is paramount, tackle complexity at the hardware level
  - Trick of micro-operations

Instruction: `add %rax, (%rcx)`

Micro-operations:

```
load  (%rcx), %r1
add %rax, %r1
store %r1, (%rcx)
```

  - Instruction ⟶Micro-operation: hardware or lookup table based
  - Design of OoO is possible because of this trick

# RISC vs. CISC (Convergence)

- CISC simplifications (enhancements)
  - Additional registers
  - Registers for floating point
  - Segmentation
  - Context switch
- RISC adaptations
  - Pseudo instructions ( li $s1, <32 bit val> ===> lui $s1, <16bit MSB>; ori $s1, <16 bit LSB>)
  - Privilege architecture: Expanded modes of privilege (risc-v)