Grader and Prerequisite

- Chinmay Kulkarni
- Email: chinmayk93@gmail.com
- Lab sessions: Monday&Wednesday 4 pm ~ 5 pm at GMCS 557
- Please send an Hello email to him: email subject is "Hello, cs572!", in the email body, please tell him your Red ID, full name, your working email address if it is not the one in the Hello email, and a screenshot of your transcript to show that you passed (D or above) the prerequisite of cs572 (i.e., CS 370 or equivalent).

Review

- Two performance metrics execution time and throughput.
- Measuring CPU time: CPI

CPU time = Instruction count x CPI x clock cycle time

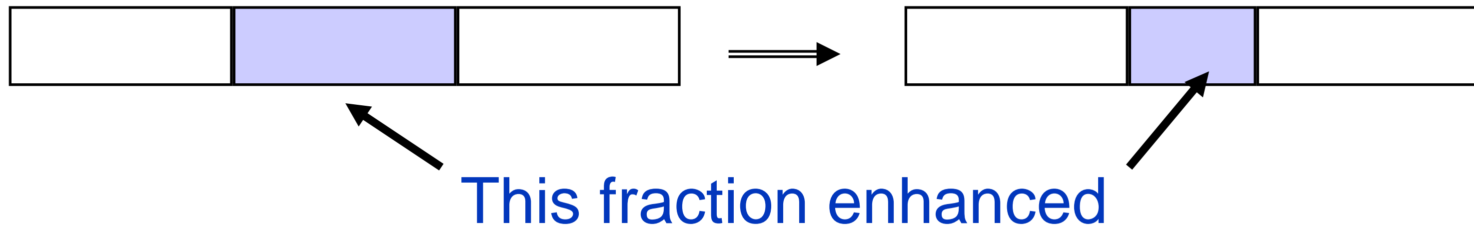
CPU time = Instruction count x CPI / clock rate

Quantitative Design: Amdahl's Law

Amdahl's Law gives a quick way to find the speedup from some enhancement.

Speedup due to enhancement E:

$$Speedup(E) = \frac{Execution_Time_Without_Enhancement}{Execution_Time_With_Enhancement} = \frac{Performance_With_Enhancement}{Performance_Without_Enhancement}$$

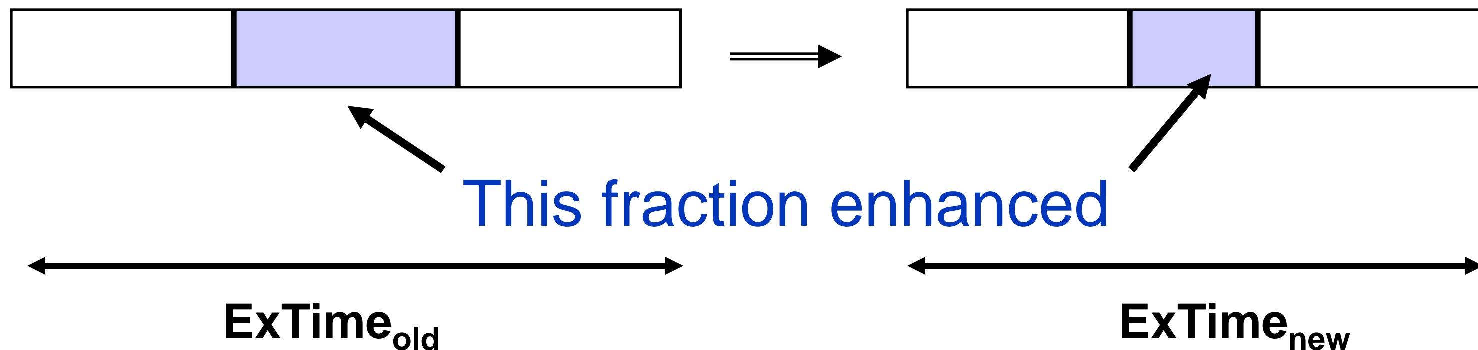


Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

Quantitative Design: Amdahl's Law

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

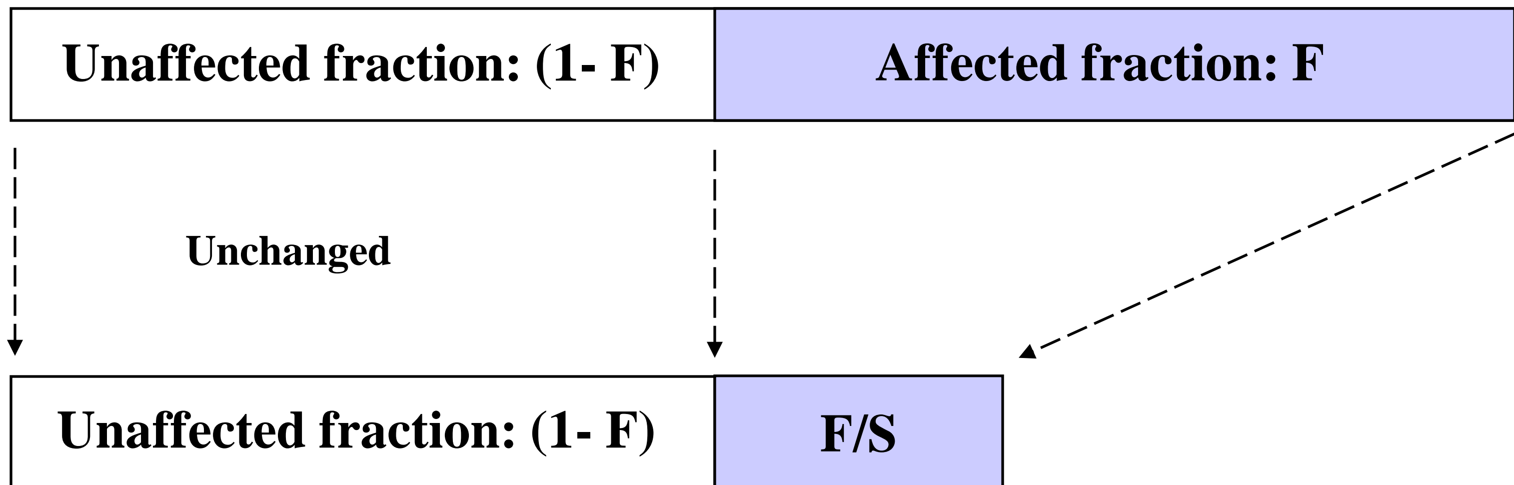


Pictorial Depiction of Amdahl's Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before: Execution Time without enhancement E

- shown normalized to 1 = (1-F) + F =1



After: Execution Time with enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

Quantitative Design: Amdahl's Law

- Floating point (FP) instructions improved to run 2X; but only 10% of actual instructions are FP. Suppose the old execution time is $ExTime_{old}$, What are the current execution time and speedup?

$$ExTime_{new} = ExTime_{old} \times (0.9 + 0.1/2) = ExTime_{old} \times 0.95$$

$$Speedup_{overall} = \frac{1}{0.95} = 1.053$$

$$Speedup = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$Speedup = \frac{1}{(1 - 0.1) + 0.1/2} = 1.053$$

Performance Summary

- Two performance metrics execution time and throughput.
- Measuring CPU time: CPI

  CPU time = Instruction count x CPI x clock cycle time

  CPU time = Instruction count x CPI / clock rate

- Amdahl's Law

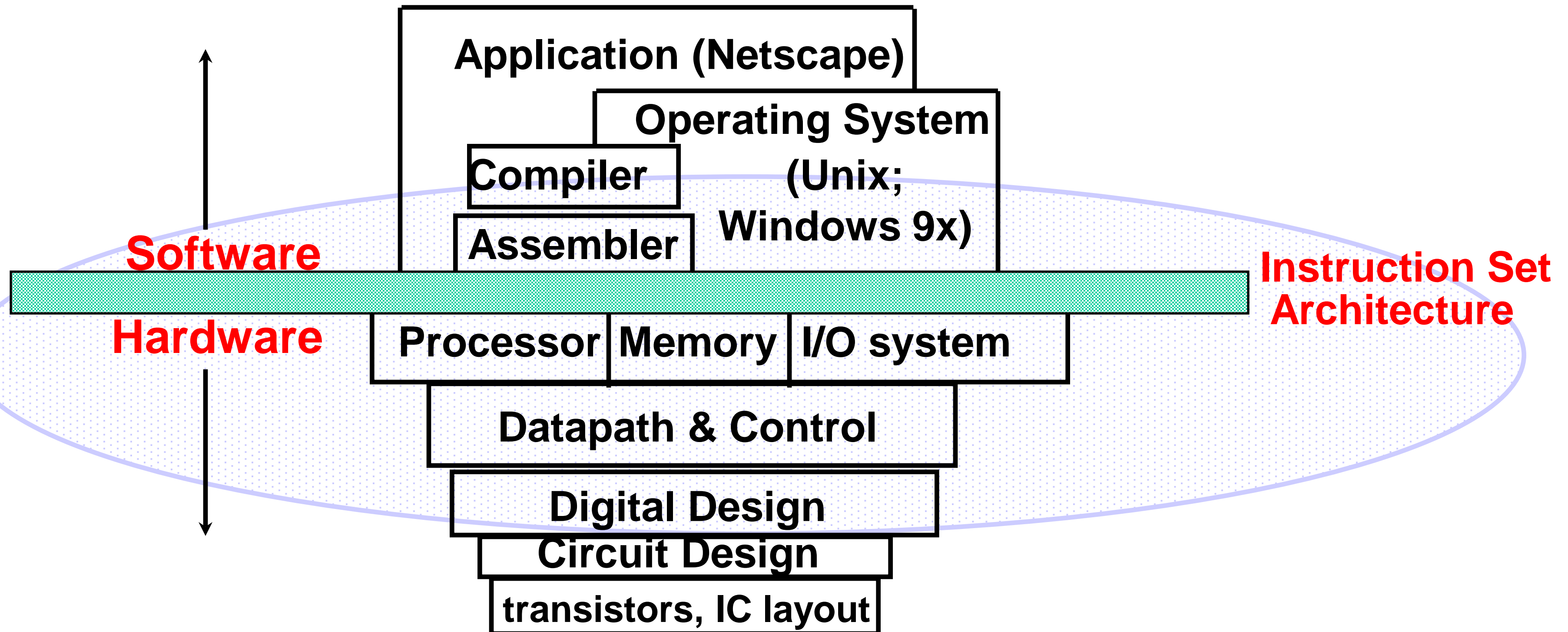
$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

- When trying to improve performance, look at what occurs frequently => make the common case fast.

Outline

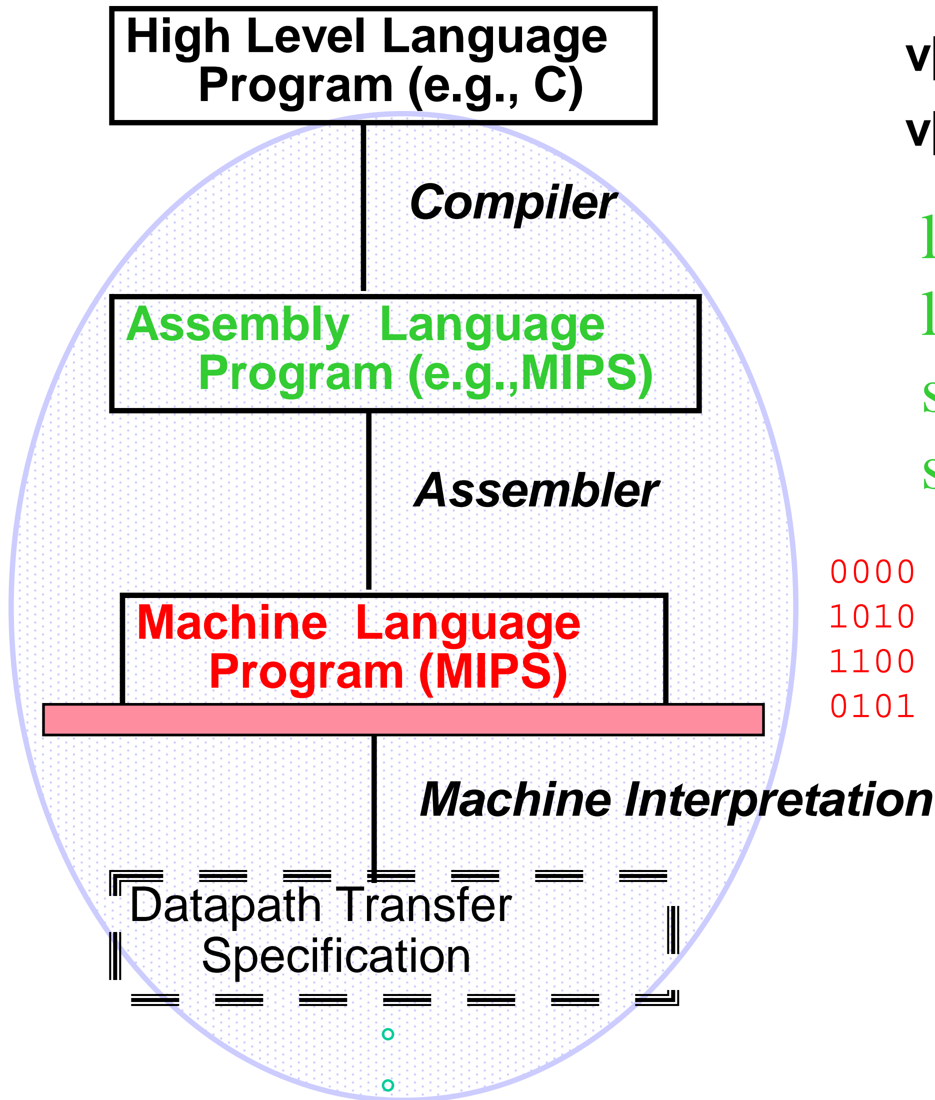
- Instruction Set Overview
  - Classifying Instruction Set Architectures (ISAs)
  - Memory Addressing
  - Types of Instructions

Instruction Set Architecture (ISA)



- Serve as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.

Instruction Set Architecture (ISA)



temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

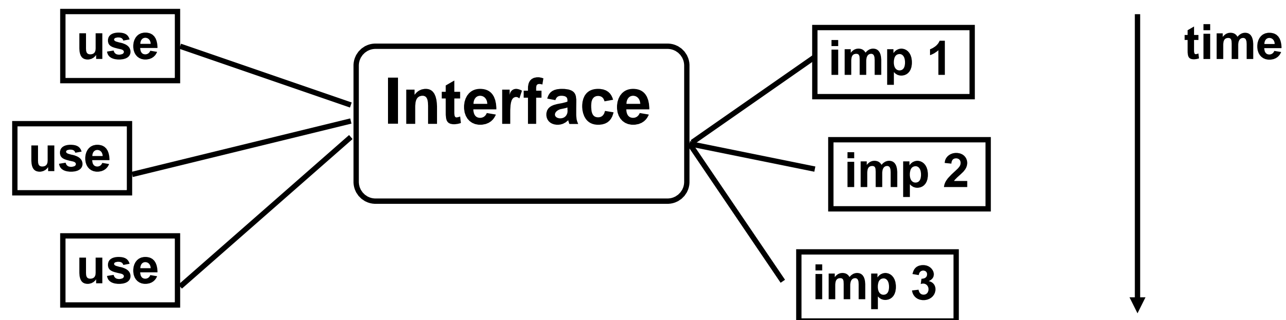
```
lw   $15,  0($2)
lw   $16,  4($2)
sw   $16,  0($2)
sw   $15,  4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

IR <– Imem[PC]; PC <– PC + 4

Interface Design

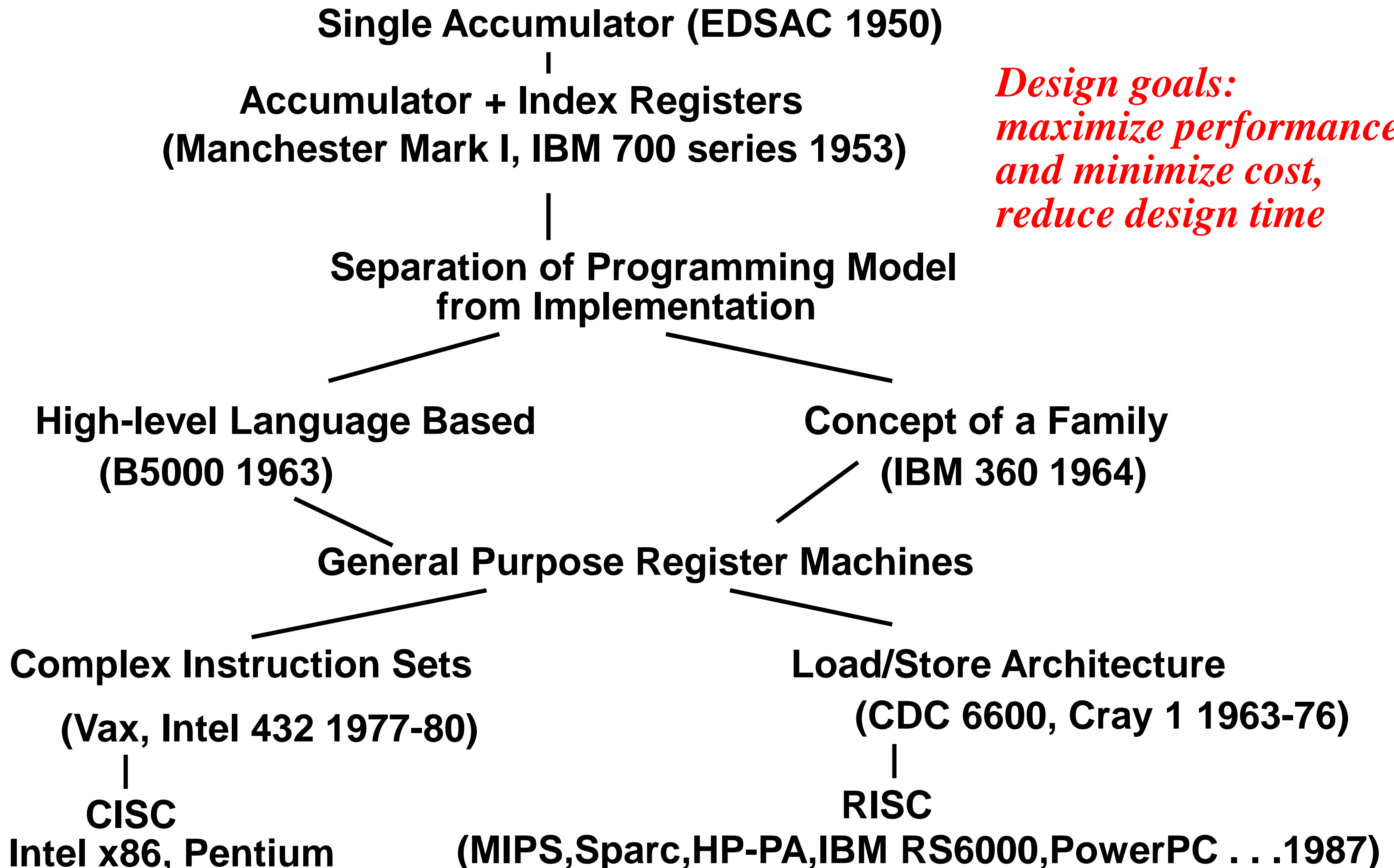
- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels



Instruction Set Design Issues

- Where are operands stored?
  - registers, memory, stack, accumulator
- How many explicit operands are there?
  - 0, 1, 2, or 3
- How is the operand location specified?
  - register, immediate, indirect, . . .
- What type & size of operands are supported?
  - byte, int, float, double, string, vector. . .
- What operations are supported?
  - add, sub, mul, move, compare . . .

Evolution of Instruction Sets



Classifying ISAs

Accumulator (1 register):

1 address   add A   acc ← acc + mem[A]

1+x address   addx A   acc ← acc + mem[A + x]

Stack:

0 address   add   tos ← tos + next

General Purpose Register:

2 address   add A B   EA(A) ← EA(A) + EA(B)

3 address   add A B C   EA(A) ← EA(B) + EA(C)

Load/Store:

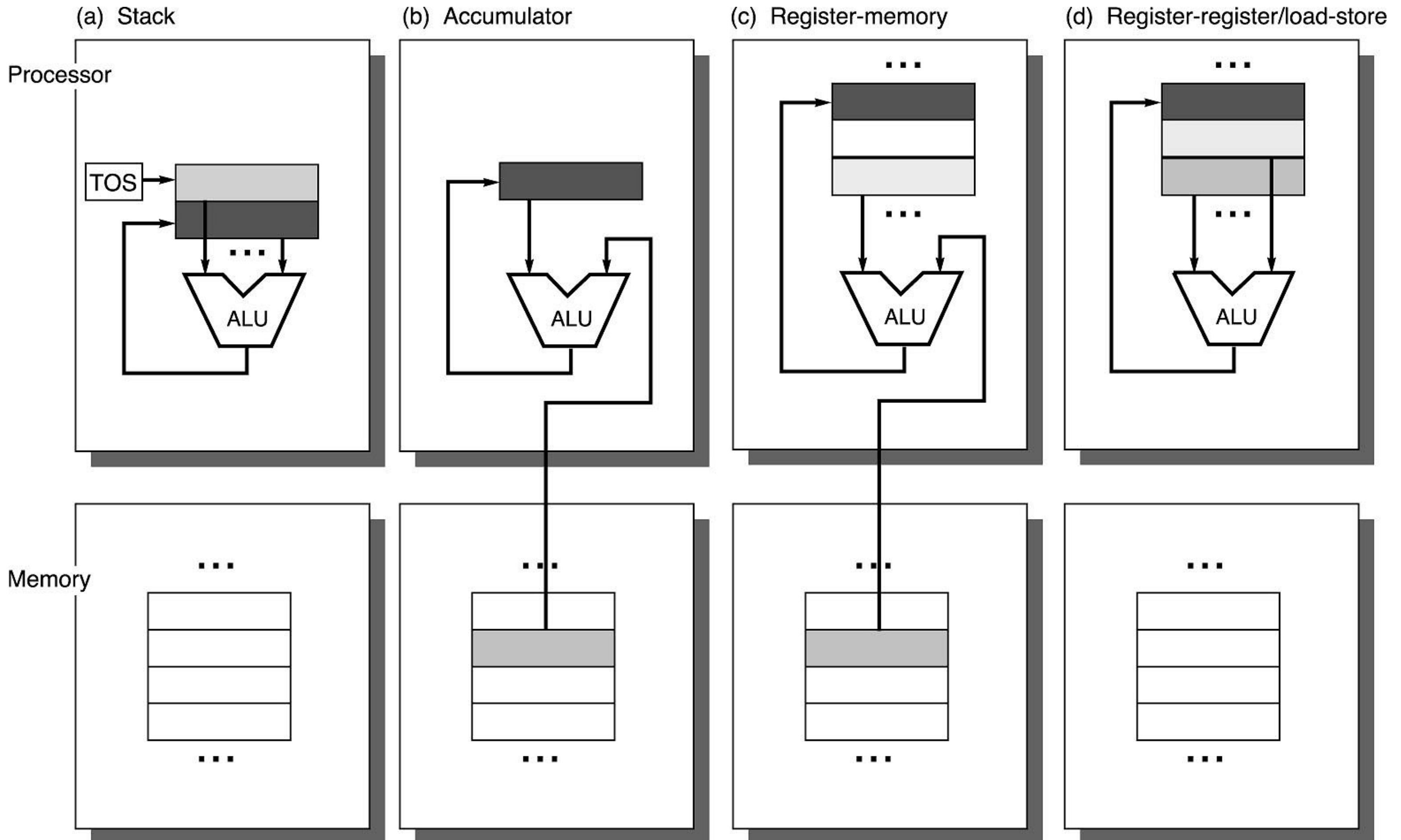
load Ra Rb   Ra ← mem[Rb]

store Ra Rb   mem[Rb] ← Ra

Memory to Memory:

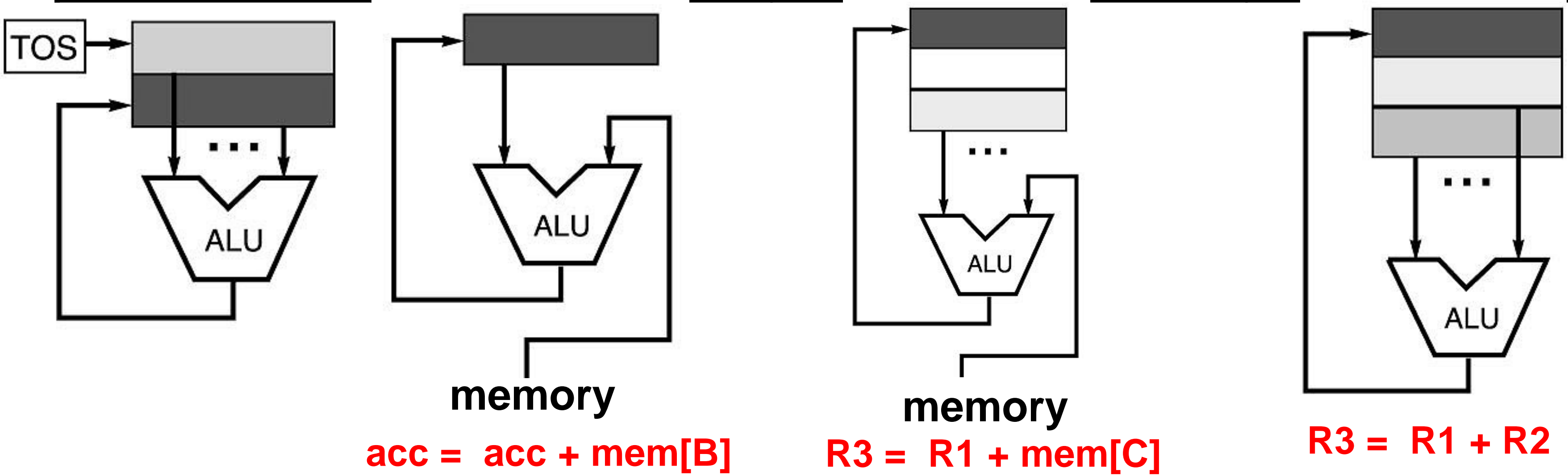
All operands and destinations can be memory addresses.

Operand Locations in Four ISA Classes



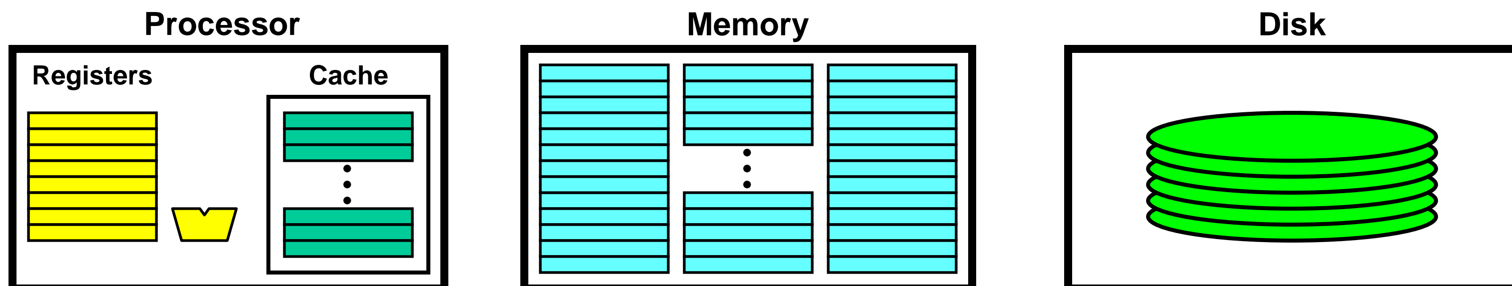
Code Sequence C = A + B for Four Instruction Sets

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R3, R1, B Store R3, C	Load R1,A Load R2, B Add R3, R1, R2 Store R3, C



General Purpose Registers (GPR)

- Why GPRs Dominate?
  - Registers are much faster than memory (even cache)
    - Register values are available immediately
    - When memory isn't ready, processor must wait ("stall")
  - Registers are convenient for variable storage
    - Compiler assigns some variables just to registers
    - More compact code since small fields specify registers (compared to memory addresses)

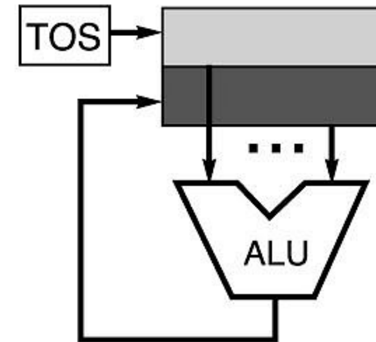


Stack Architectures

- Instruction set:

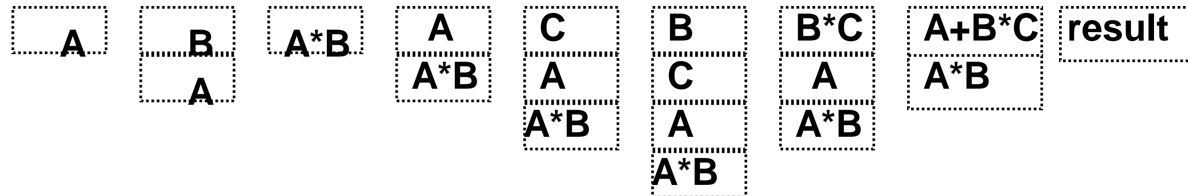
  add, sub, mul, div, . . .

  push A, pop A



- Example: A*B - (A+C*B)

  push A
  push B
  mul
  push A
  push C
  push B
  mul
  add
  sub



Stacks: Pros and Cons

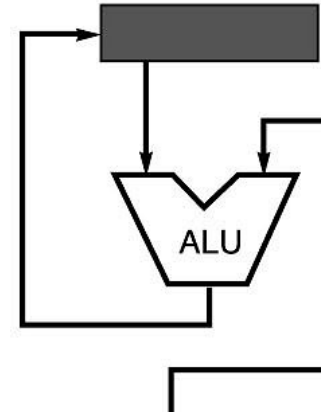
- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

Accumulator Architectures

- Instruction set:

  add A, sub A, mul A, div A, . . .

  load A, store A



acc = acc +,-,*,/ mem[A]

- Example: A*B - (A+C*B)

  load B

  mul C

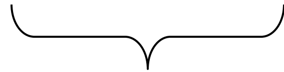
  add A

  store D

  load A

  mul B

  sub D



mem[D]

Accumulators: Pros and Cons

- Pros
  - Very low hardware requirements
  - Easy to design and understand

- Cons
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Memory-Memory Architectures

- Instruction set:

(3 operands)	add A, B, C	sub A, B, C	mul A, B, C
(2 operands)	add A, B	sub A, B	mul A, B

- Example: A*B - (A+C*B)

  A*B → mem[D]; (A+C*B) → mem[E]

- 3 operands

```
mul D, A, B
mul E, C, B
add E, A, E
sub E, D, E
```

2 operands

```
mov D, A
mul D, B
mov E, C
mul E, B
add E, A
sub E, D
```

Memory-Memory: Pros and Cons

- Pros
  - Requires fewer instructions (especially if 3 operands)
  - Easy to write compilers for (especially if 3 operands)
- Cons
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

Register-Memory Architectures

- Instruction set:

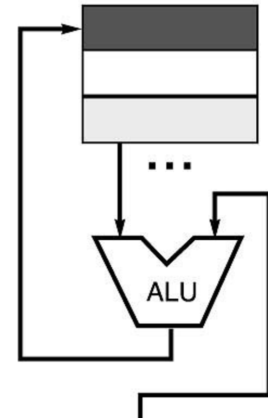
  add R1, A   sub R1, A   mul R1, B

  load R1, A   store R1, A

- Example: A*B - (A+C*B)

  mem[D] (A*B)   R2 (A+C*B)

```
load R1, A
mul R1, B       /*    A*B             */
store R1, D
load R2, C
mul R2, B       /*    C*B             */
add R2, A       /*    A + CB          */
sub R2, D       /*    AB - (A + C*B)  */
```



R1 = R1 +,-,*,/ mem[B]

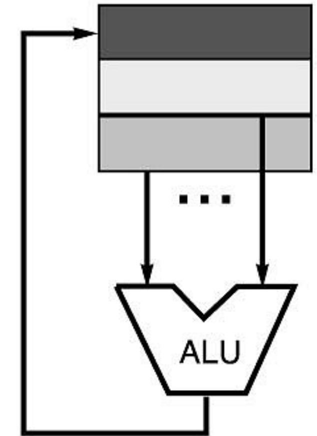
Memory-Register: Pros and Cons

- Pros
  - Some data can be accessed without loading first
  - Instruction format easy to encode
  - Good code density
- Cons
  - Operands are not equivalent
  - Variable number of clocks per instruction
  - Limit number of registers

Load-Store Architectures

- Instruction set:

  add R1, R2, R3   sub R1, R2, R3   mul R1, R2, R3
  load R1, &A   store R1, &A   move R1, R2



R3 = R1 +,-,*,/ R2

- Example: A*B - (A+C*B)

  R9   R7   R8   R10

  load R1, &A
  load R2, &B
  load R3, &C
  mul R7, R3, R2   /* C*B */
  add R8, R7, R1   /* A + C*B */
  mul R9, R1, R2   /* A*B */
  sub R10, R9, R8   /* A*B - (A+C*B) */

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar
- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Registers:Advantages and Disadvantages

- Advantages
  - Faster than cache or main memory (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic
- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number
- ISAs designed after 1980 use a load-store ISA (i.e MIPS,Sparc, HP-PA,IBM RS6000,PowerPC, to simplify CPU design).

Summary

- Instruction Set Overview
  – Classifying Instruction Set Architectures (ISAs)

For Next Time…

- Memory Addressing
- Types of Instructions
- MIPS Instruction Set

The grader will email you a supplemental reading document after you send him a Hello email. You need to finish the reading by Sept. 10.