KING'S College LONDON

Hardware Design

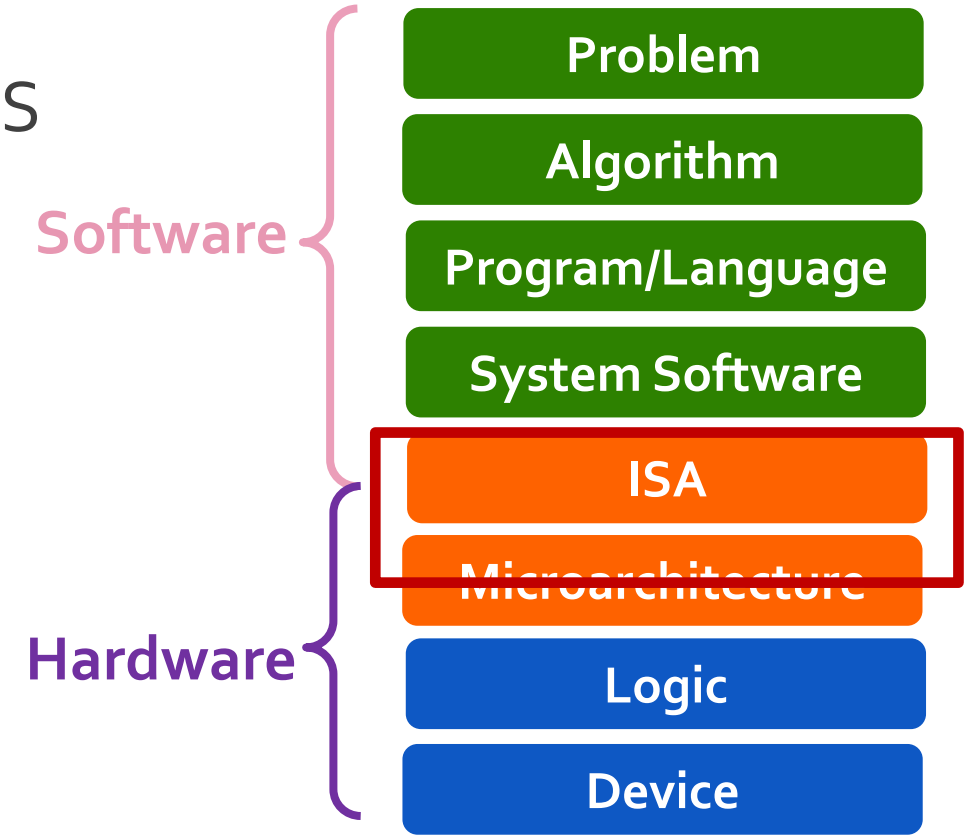
Lecture 3: ISA (III), Microarchitecture, and Assembly

Dr. Haiyu Mao

05.02.2026

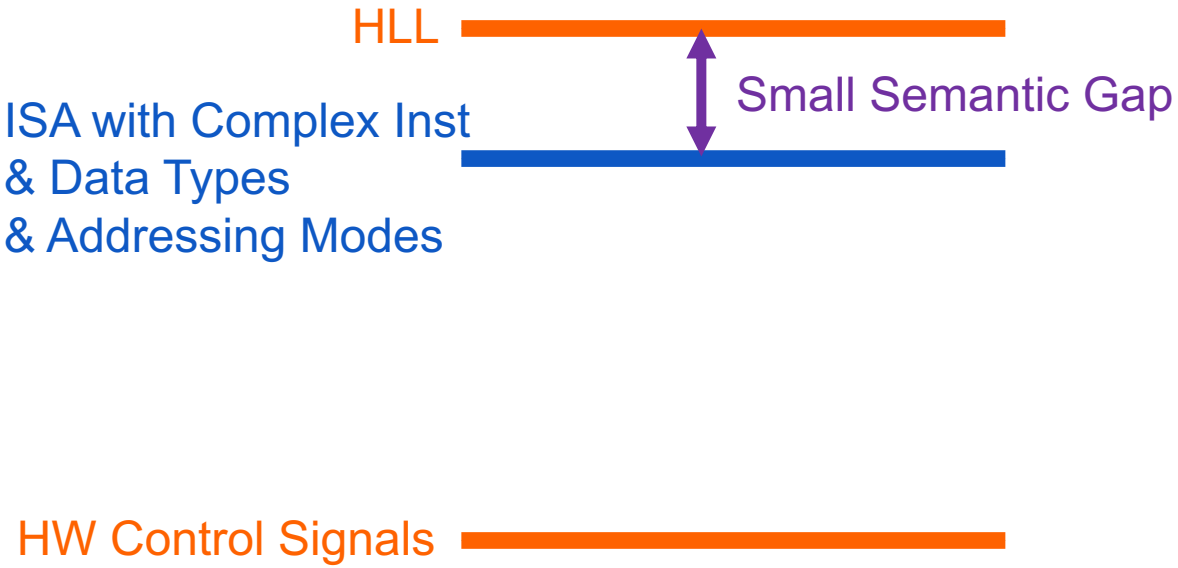
What We Learned

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine
- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions
- Instruction formats
- Data types & Addressing modes

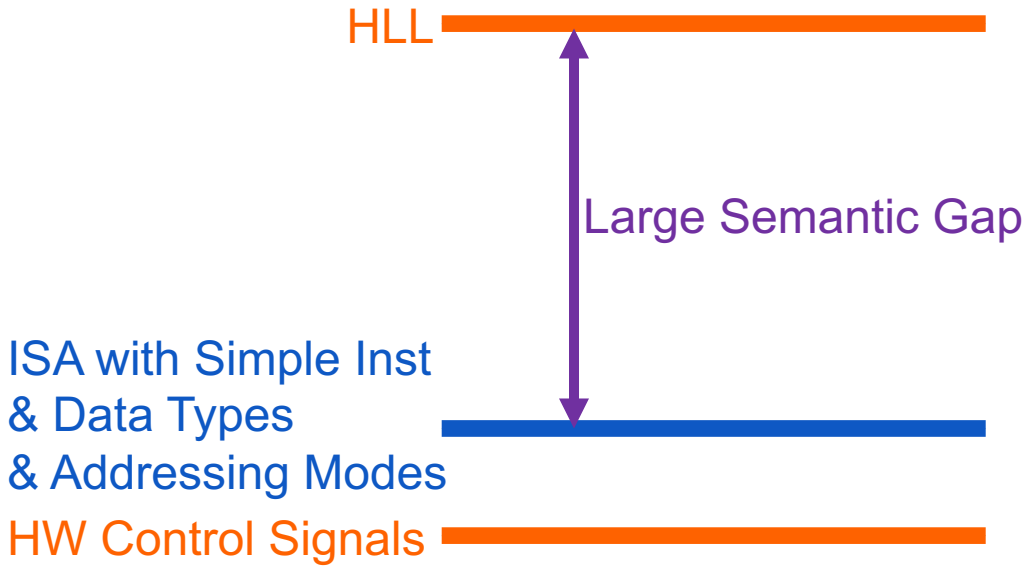


Recall: Semantic Gap

- How close are instructions & data types & addressing modes to high-level language (HLL)



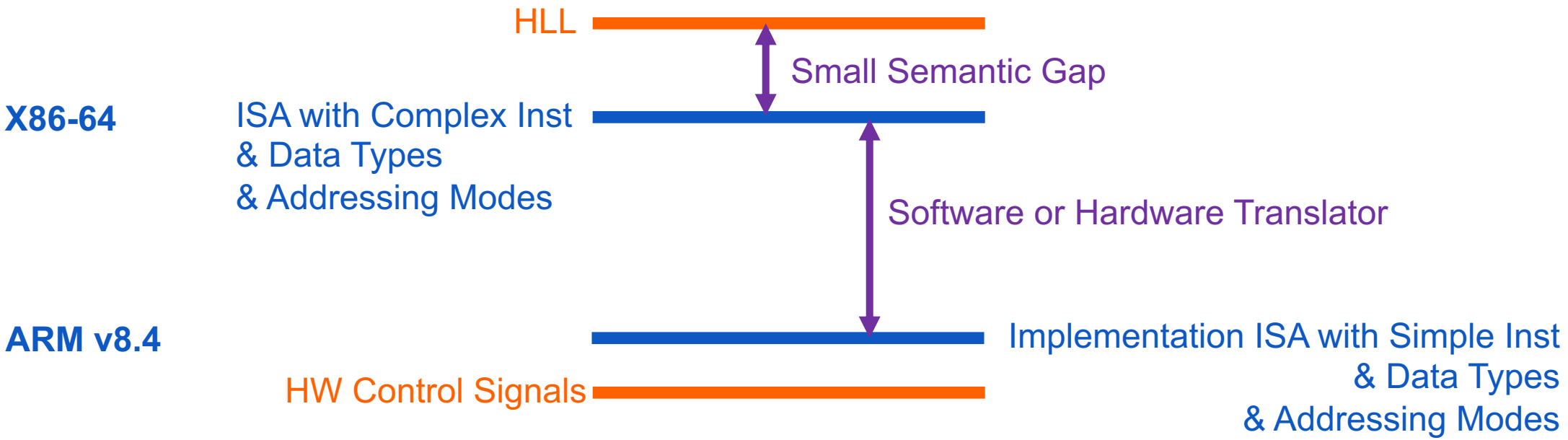
Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW



Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW

How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different "implementation" ISA



An Example: Rosetta 2 Binary Translator

Rosetta 2 [edit]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Mac transition to Apple silicon

Apple silicon · ARM architecture · Universal 2 binary · Rosetta 2 · Developer Transition Kit

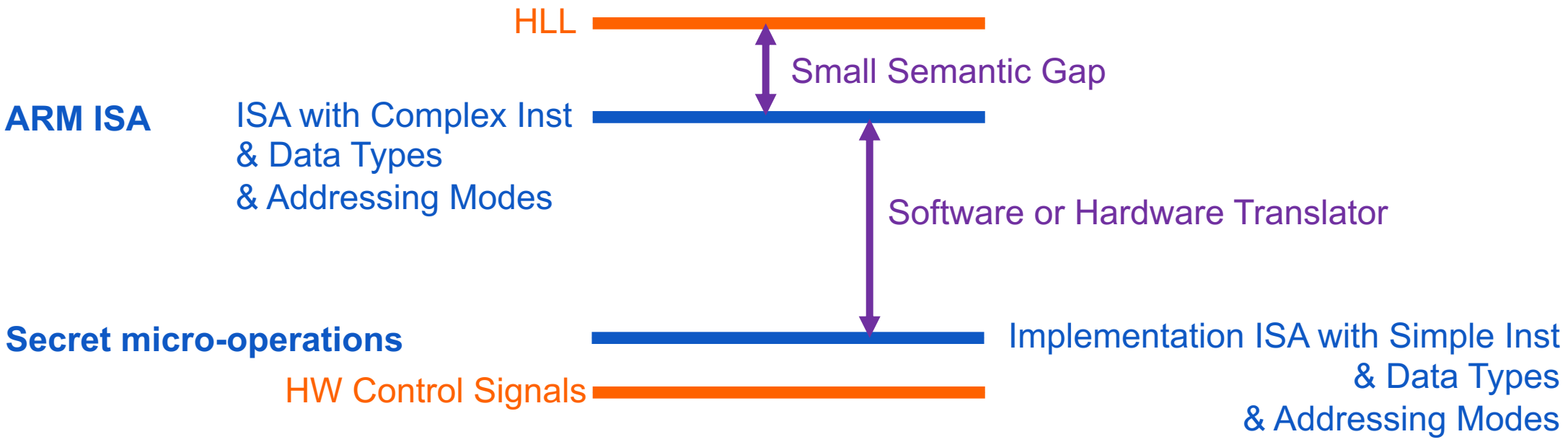
V · T · E

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]

Another Example: NVIDIA Denver

- Translate from one ISA into a different "implementation" ISA

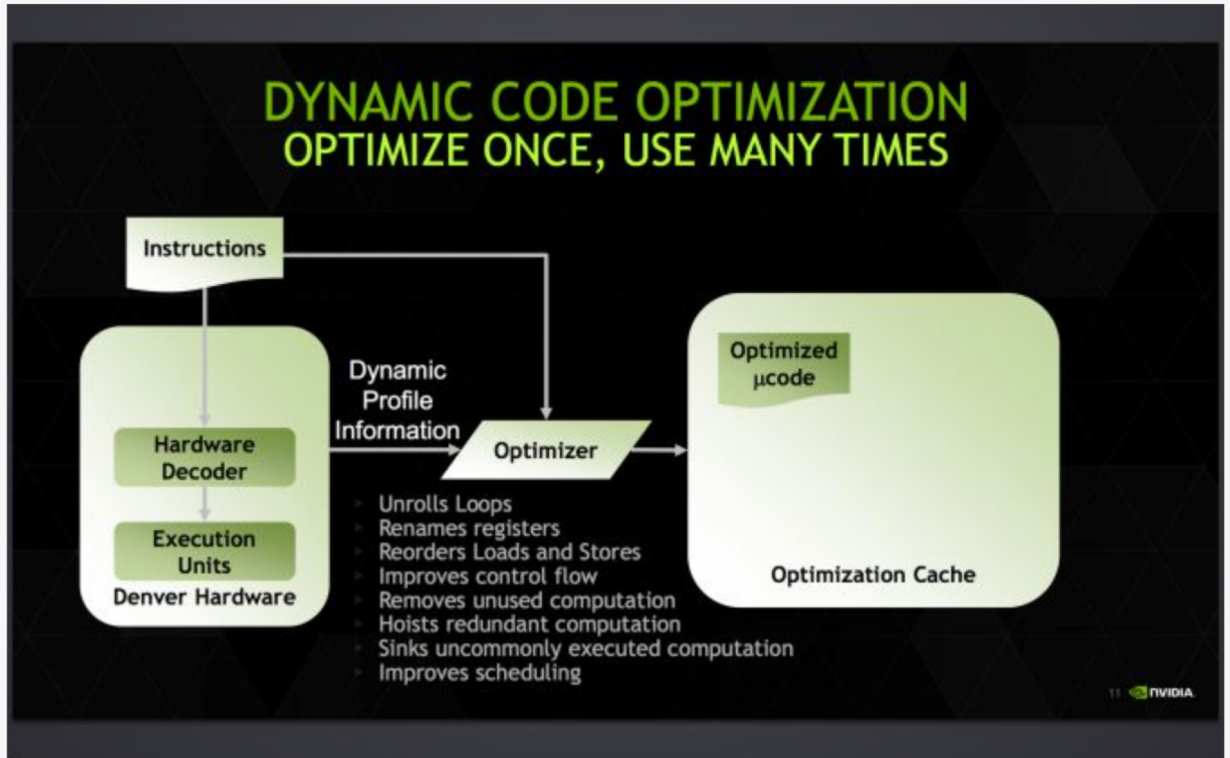


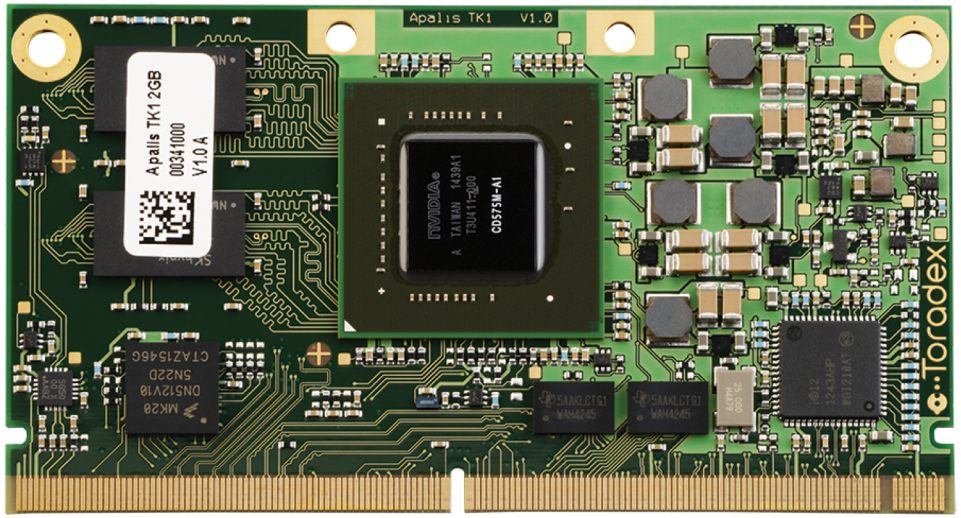
Another Example: NVIDIA Denver

The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.





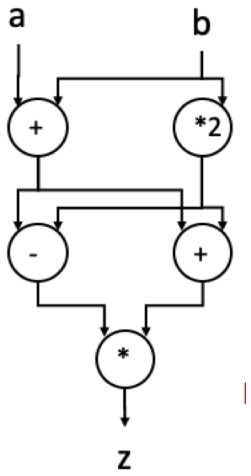
Principle: Indirection

- "Any problem in computer science can be solved with another level of indirection."
  - David Wheeler
  - Attributed in Butler Lampson's "Principles for Computer Systems Design" https://arxiv.org/pdf/2011.02455

- Tradeoffs can change, and new opportunities become enabled once you add a level of indirection
- But indirection comes with extra complexity and latency

Hardware Design

Dataflow Model of A Computer



The Dataflow Model (of a Computer)

- von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the program counter (instruction pointer)
  - Sequential unless an explicit control flow instruction

- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no program counter (instruction pointer)
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received
  - Potentially, many instructions can execute at the same time
    - Inherently more parallel

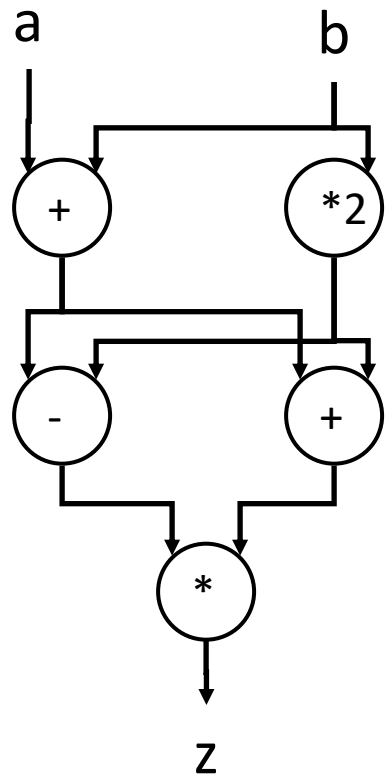
Von Neumann vs. Dataflow

- Consider a Von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y
```

Sequential

a, b are the only inputs
z is the only output



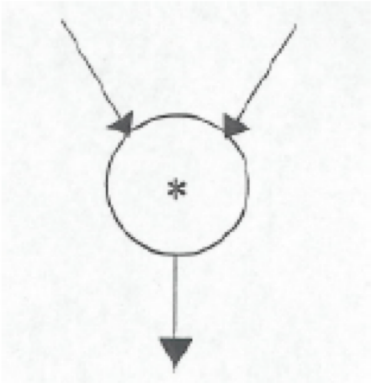
Dataflow

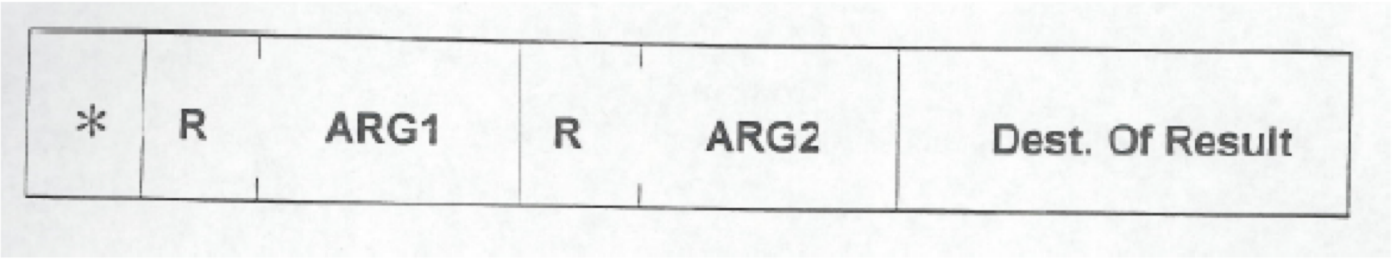
Which model is more natural to you as a programmer?

More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
  - A dataflow node fires (fetched and executed) when all its inputs are ready
    - i.e., when all inputs have tokens

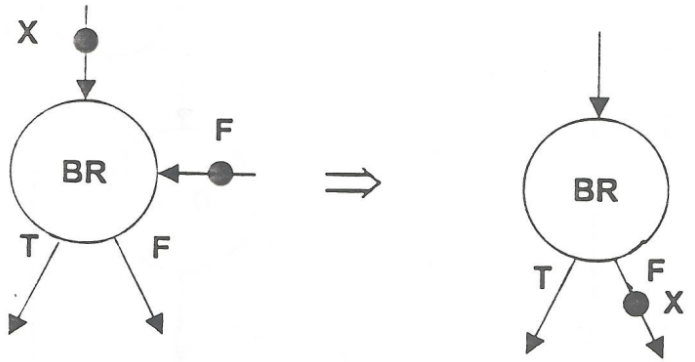
- Dataflow node and its ISA representation



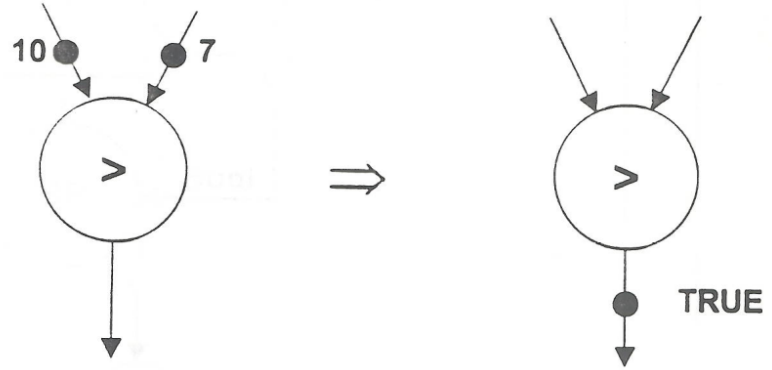


Example Dataflow Nodes

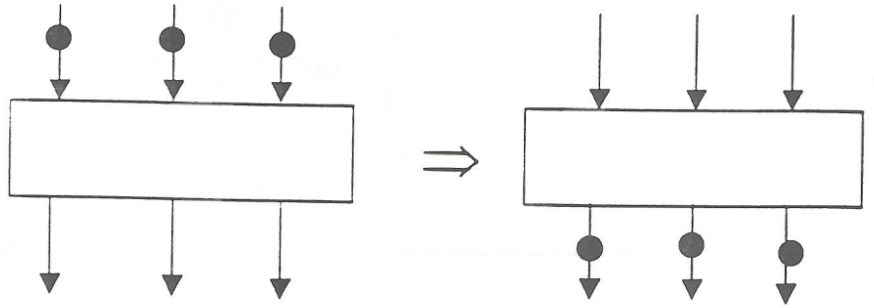
*Conditional



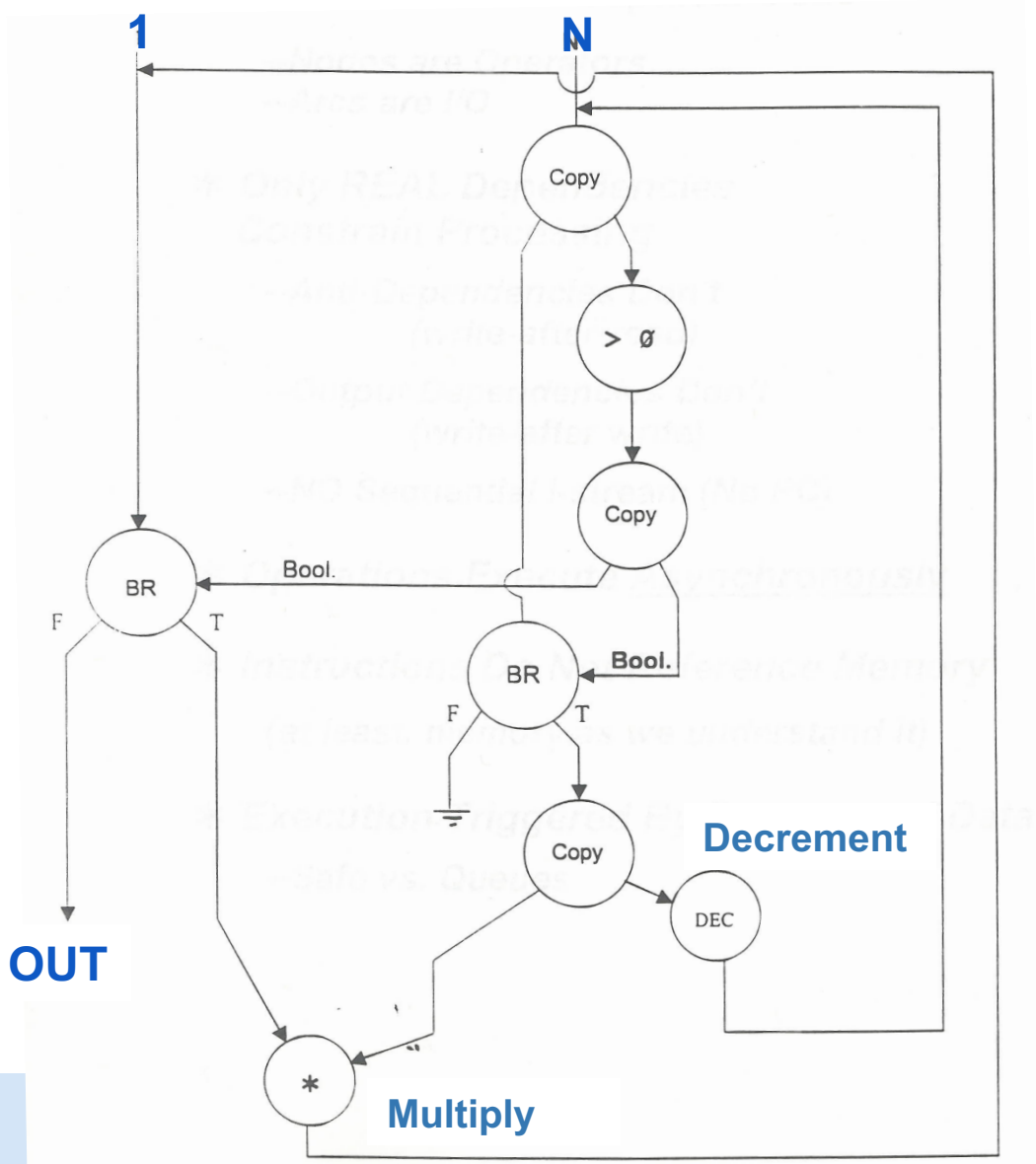
*Relational



*Barrier Synch



A Simple Example Dataflow Program



N is a non-negative integer

What is the value of OUT?

ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → sequential
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → dataflow
- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?

ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

- Microarchitecture: How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

The von Neumann Model

- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, …
- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches
- But, what happens underneath that is ***not* consistent** with the von Neumann model is ***not* exposed** to software
  - Difference between ISA and microarchitecture

Hardware Design

Computer Architecture and Microarchitecture

What is Computer Architecture?

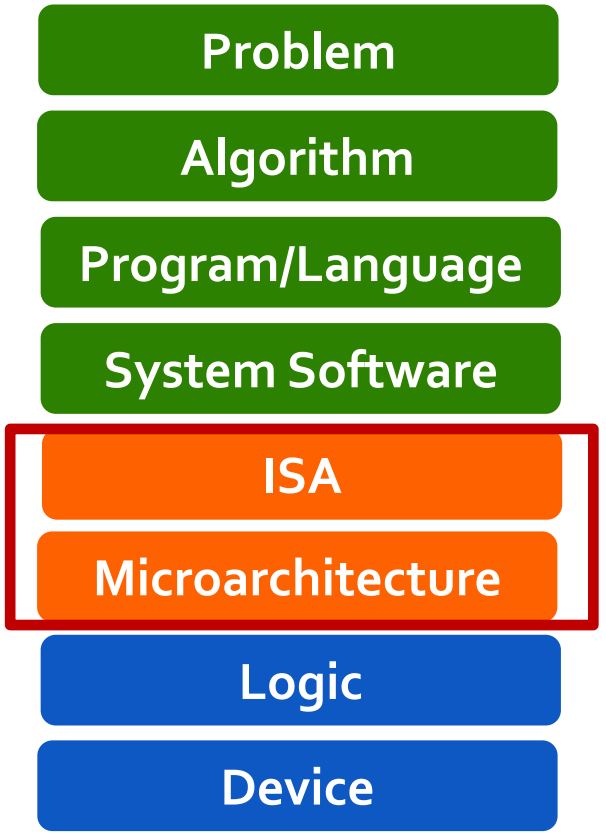
- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** "The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior ***as distinct from*** the organization of the dataflow and controls, the logic design, and the physical implementation."

  Gene Amdahl, IBM Journal of R&D, April 1964

ISA vs. Microarchitecture

- ISA
  - Agreed upon the interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Computer Architecture" = ISA + microarchitecture



Microarchitecture

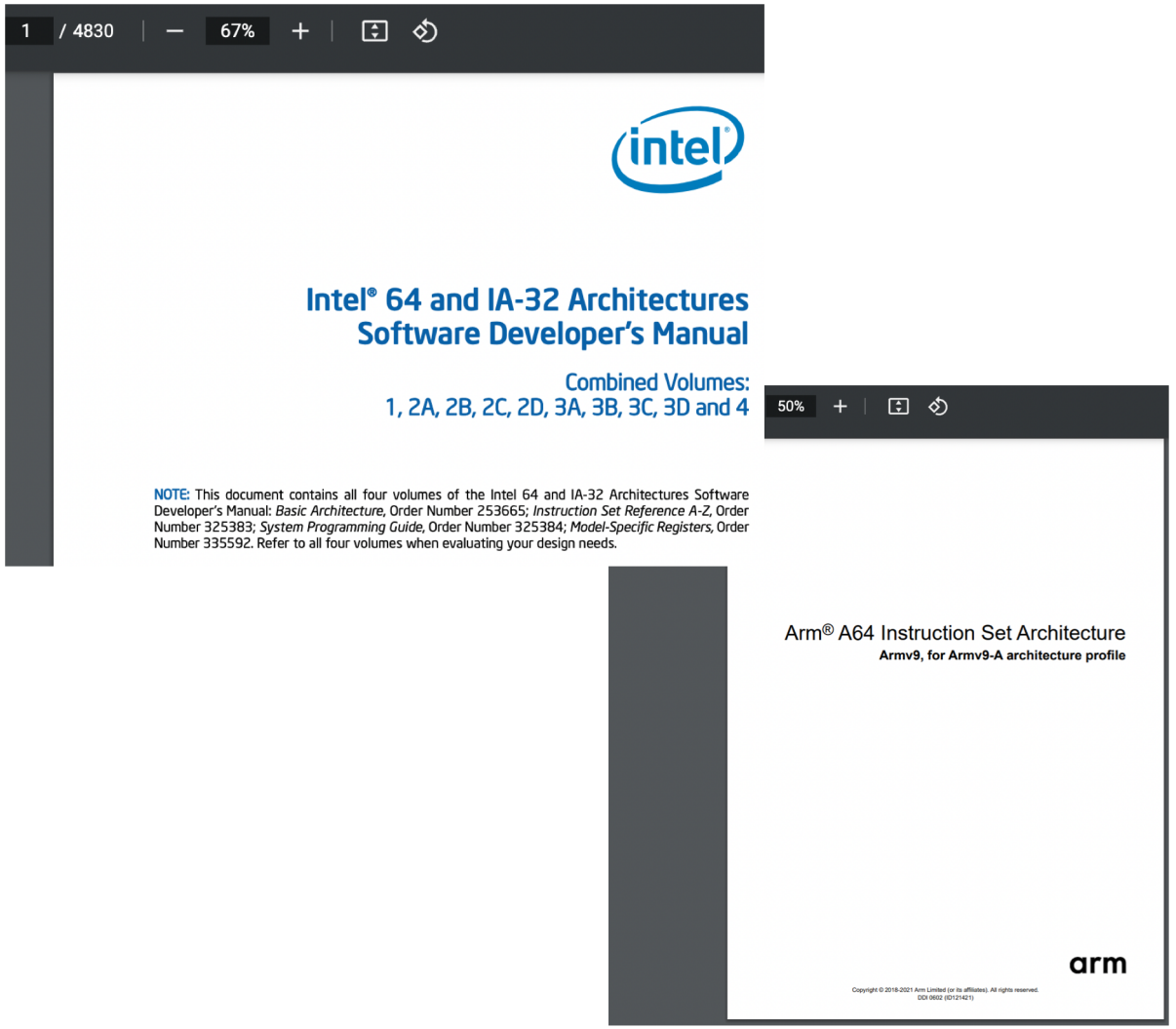
- A specific **implementation** of the ISA
- How do we implement the ISA?
  - We will discuss this for several lectures
- There can be many implementations of the same ISA
  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, …
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, …, AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, …
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), …, **PowerPC** 604, 605, 620, …
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, …
  - **Alpha** 21064, 21164, 21264, 21364, …
  - **RISC-V** …
  - **z/Architecture** …
  - …

ISA vs. Microarchitecture

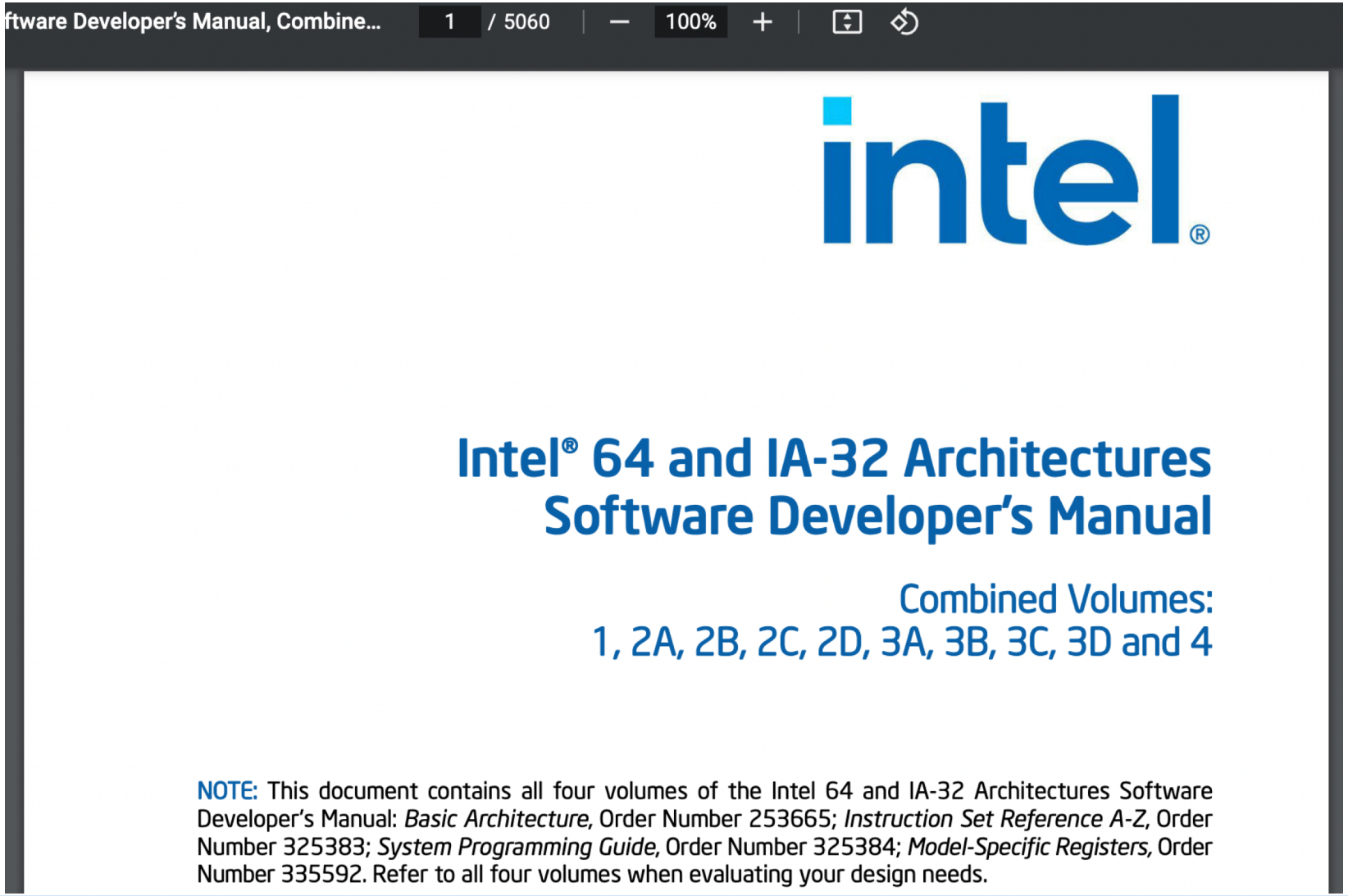
- What is part of ISA vs. Uarch?
  - Gas pedal: interface for "acceleration"
  - Internals of the engine: implement "acceleration"
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, and carry lookahead adders are all part of microarchitecture
  - x86 ISA has many implementations:
    - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cover, Sapphire Rapids, …, AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, …
- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
  - *Why?*

ISA: What Does It Specify?

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O: memory-mapped vs. instructions
- Task/thread Management
- Power & Thermal Management
- Multithreading & Multiprocessor support
- …

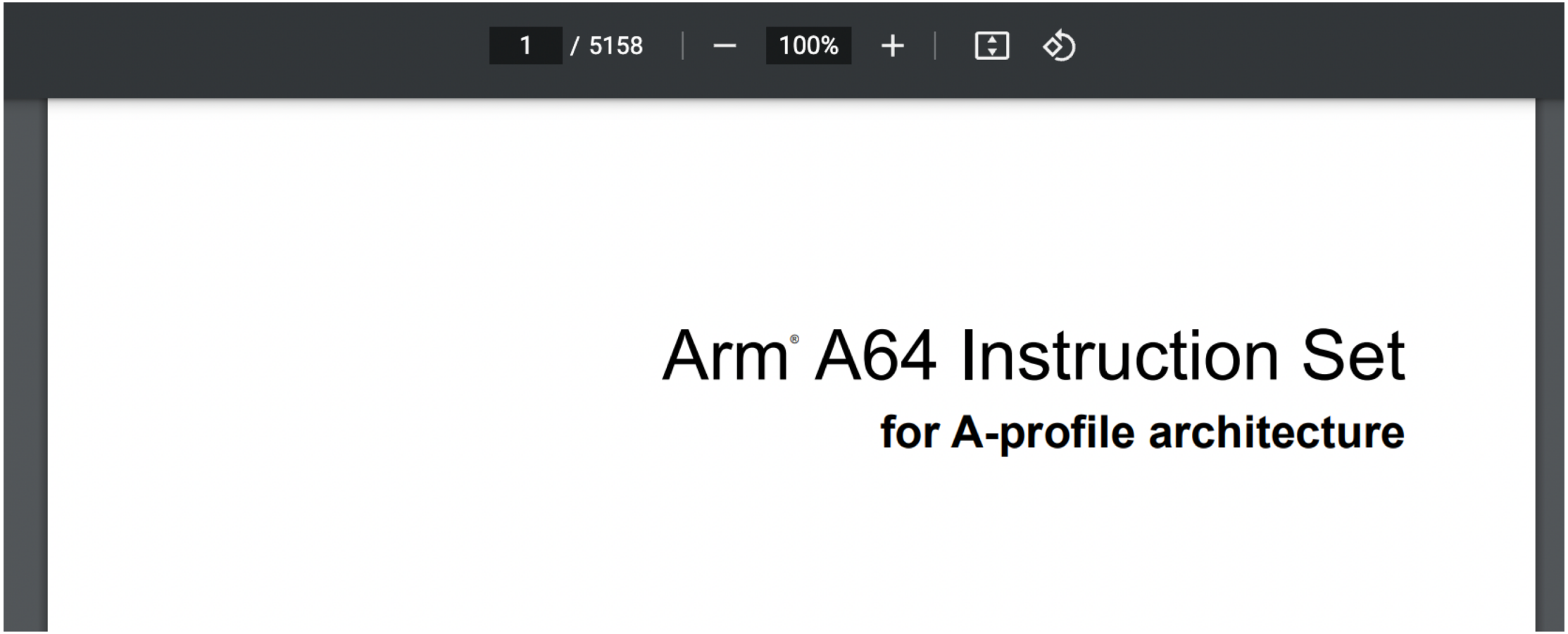


ISAs Keep Getting Extended



https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

ISAs Keep Getting Extended



https://developer.arm.com/documentation/ddi0602/latest/

Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?

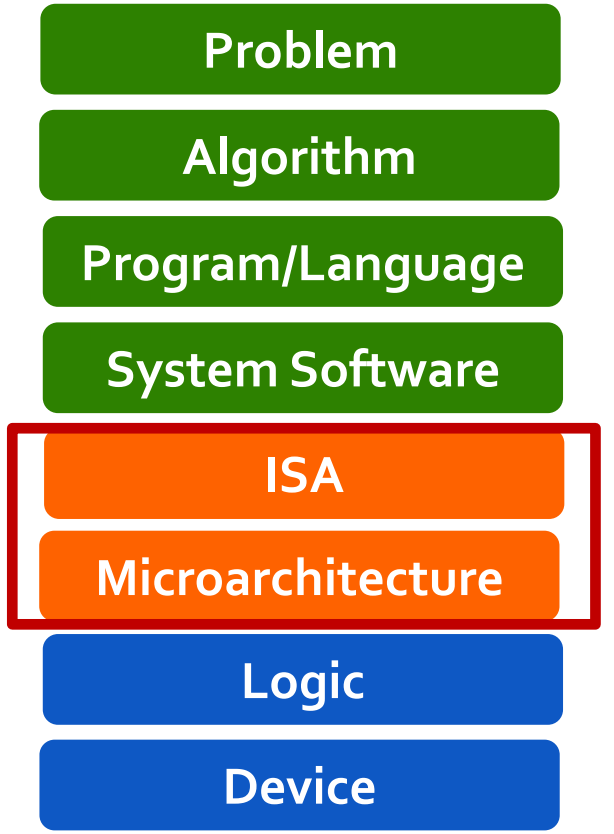
Quiz: Property of ISA vs. Microarchitecture?

- ADD instruction's opcode
- Type of adder used in the ALU (Bit-serial vs. Ripple-carry)
- Number of general-purpose registers
- Number of cycles to execute the MUL instruction
- Number of ports to the register file
- Whether or not the machine employs pipelined instruction execution
- Program counter

- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

Design Points

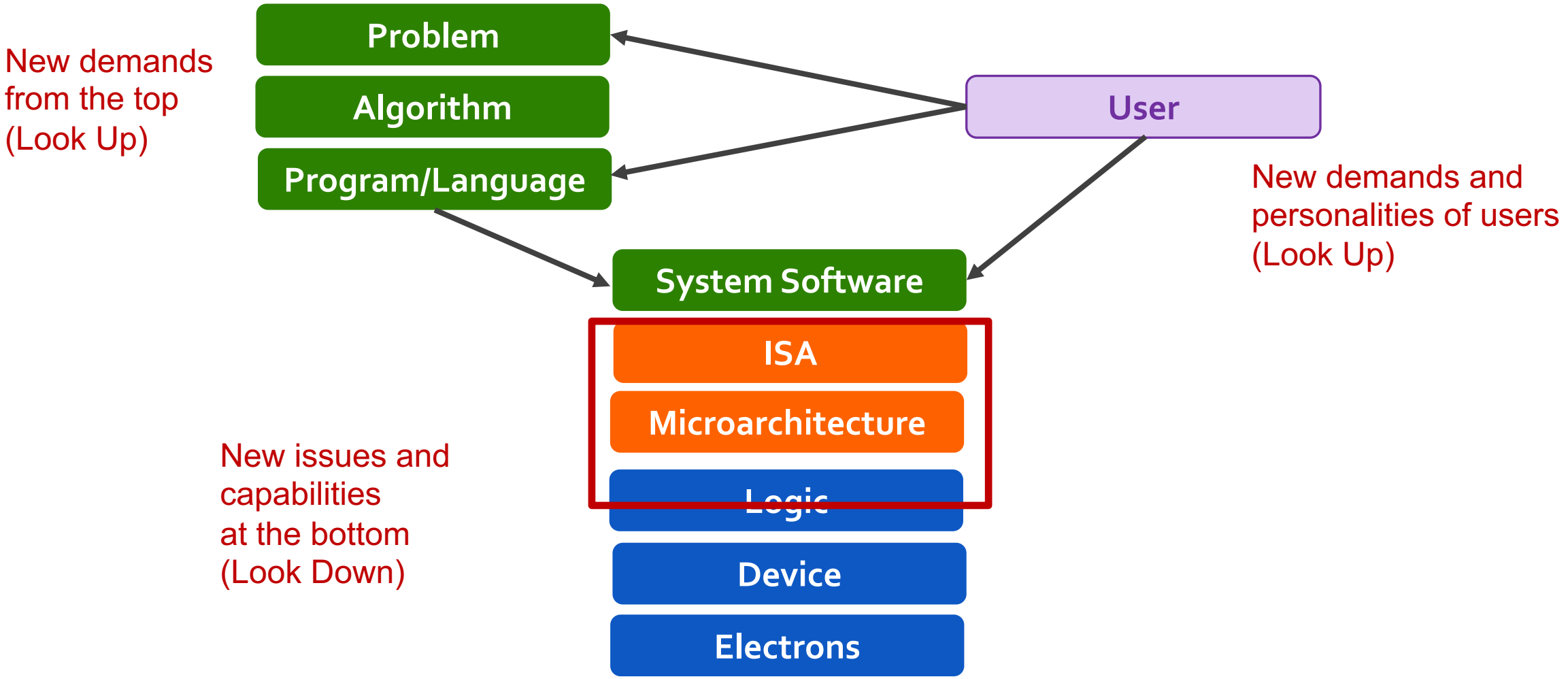
- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Example considerations:
  - Cost
  - Performance
  - Maximum power consumption, thermal
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market
  - Security, safety, predictability, …
- Design point is determined by the "Problem" space (application space), the intended users/*market*



Tradeoffs: Soul of Computer Architecture

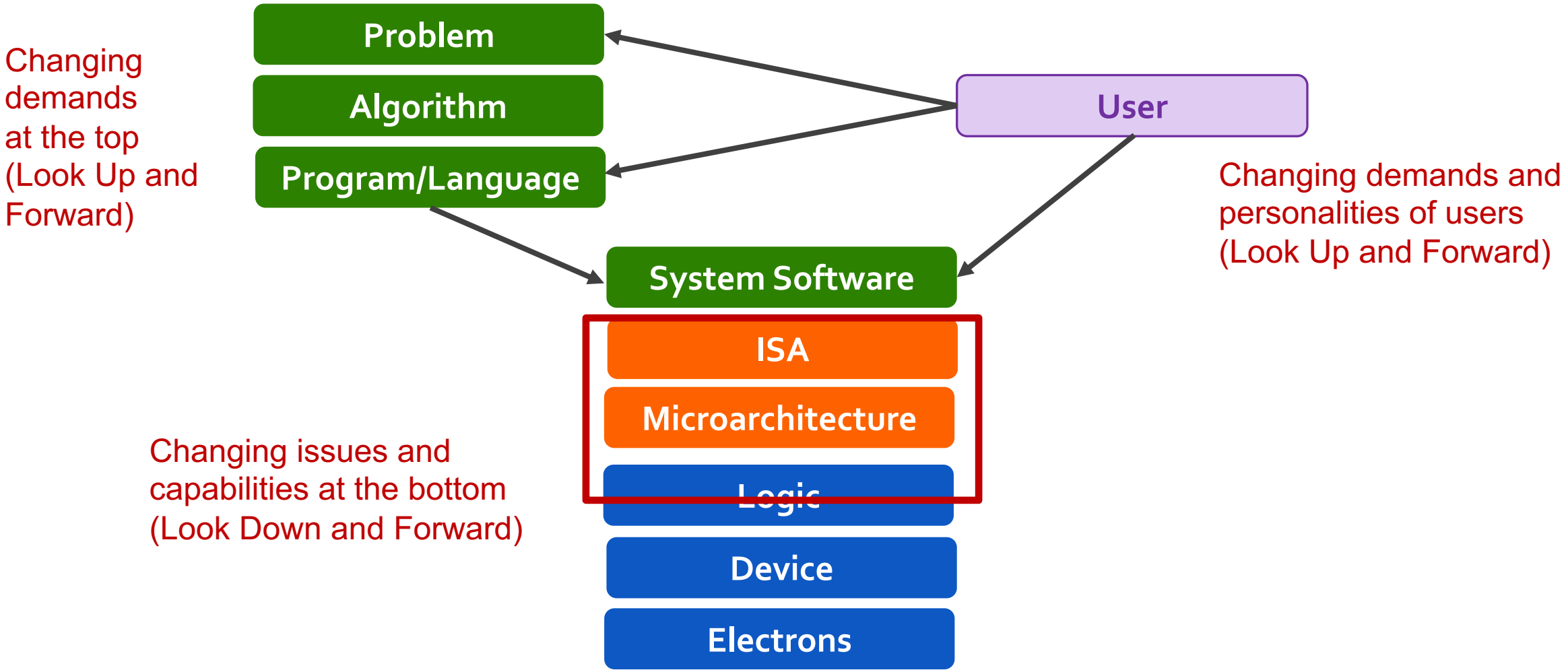
- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why **art**?*

Why Is It (Somewhat) Art?



- We do not (fully) know the future (applications, users, market)

Why Is It (Somewhat) Art?



- And, the future is not constant (it changes)!

Hardware Design

Implementing the ISA: Microarchitecture Basics

How Does a Machine Process Instructions?

- What does processing an instruction mean?
- We will assume the von Neumann model (for now)

AS = Architectural (programmer visible) state before an instruction is processed

Process instruction

AS' = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

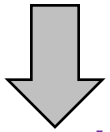
The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an **abstract finite state machine** where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From the ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have a programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

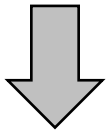
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

AS = Architectural (programmer visible) state
at the beginning of a clock cycle



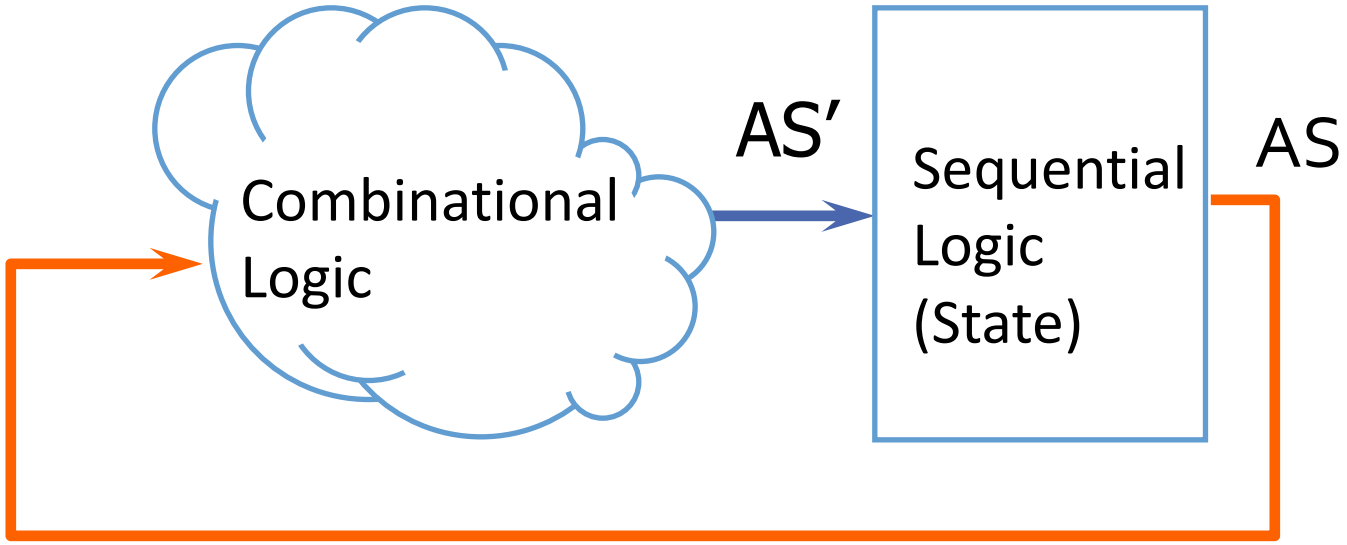
Process instruction in **one clock cycle**



AS' = Architectural (programmer visible) state
at the end of a clock cycle

A Very Basic Instruction Processing Engine

- Single-cycle machine



- What is the *clock cycle time* determined by?
- What is the *critical path* (i.e., longest delay path) of the combinational logic determined by?

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates are made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines the cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing is broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates are made at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

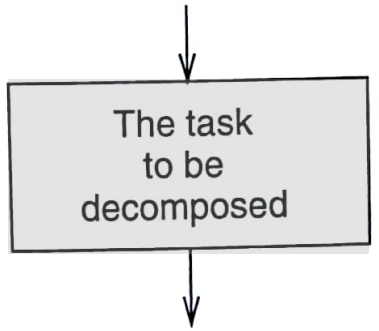
Hardware Design

Assembly Programming

```
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
```

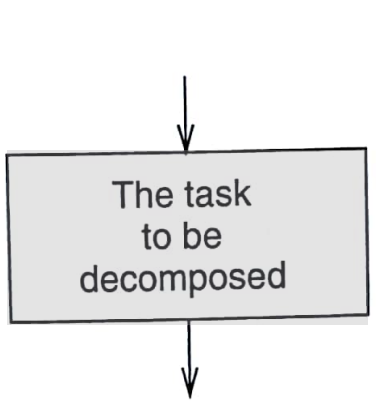
Programming Constructs

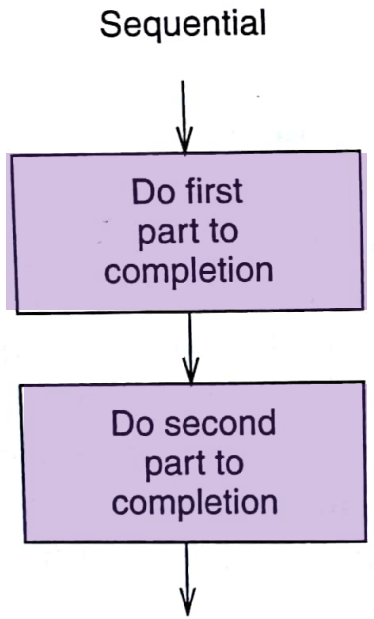
- Programming requires dividing a task, i.e., a unit of work, into smaller units of work
- The goal is to replace the units of work with programming constructs that represent that part of the task
- There are three basic programming constructs
  - Sequential construct
  - Conditional construct
  - Iterative construct



Sequential Construct

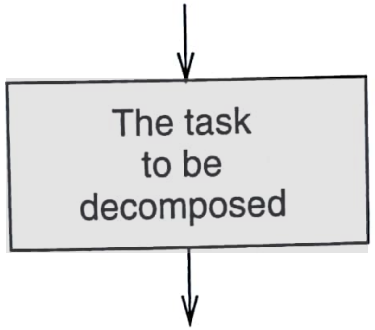
- The sequential construct is used if the designated task can be broken down into two subtasks, one following the other

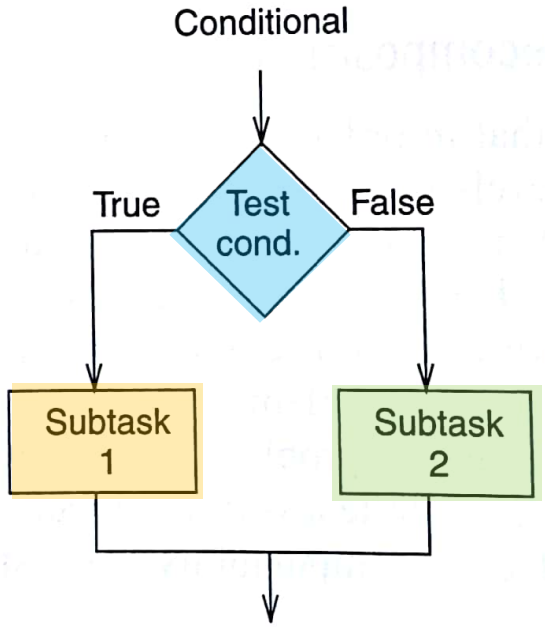




Conditional Construct

- The conditional construct is used if the designated task consists of doing one of two subtasks, but not both





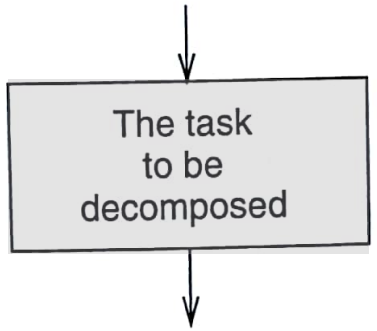
Is the condition "true" or "false"?

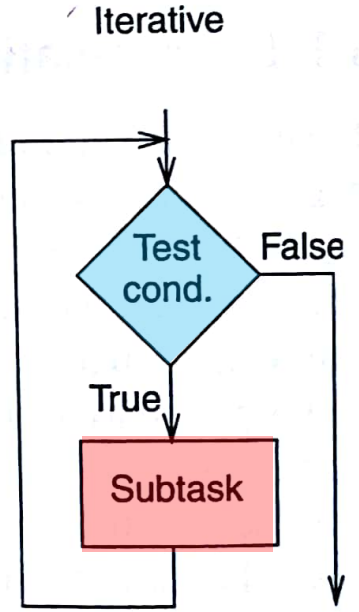
  - Either subtask may be "do nothing"
  - After the correct subtask is completed, the program moves onward

- E.g., if-else statement, switch-case statement

Iterative Construct

- The iterative construct is used if the designated task consists of doing a subtask a number of times, but only as long as some condition is true





Is the condition still "true"?

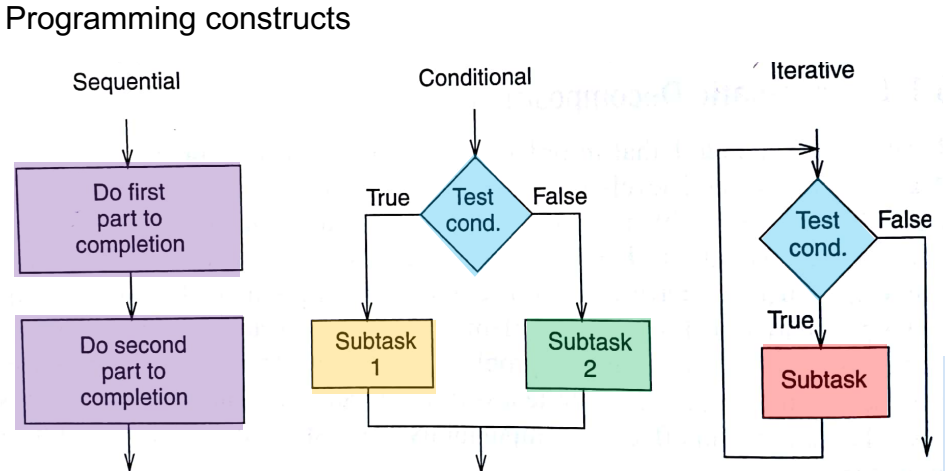
- E.g., for loop, while loop, do-while loop

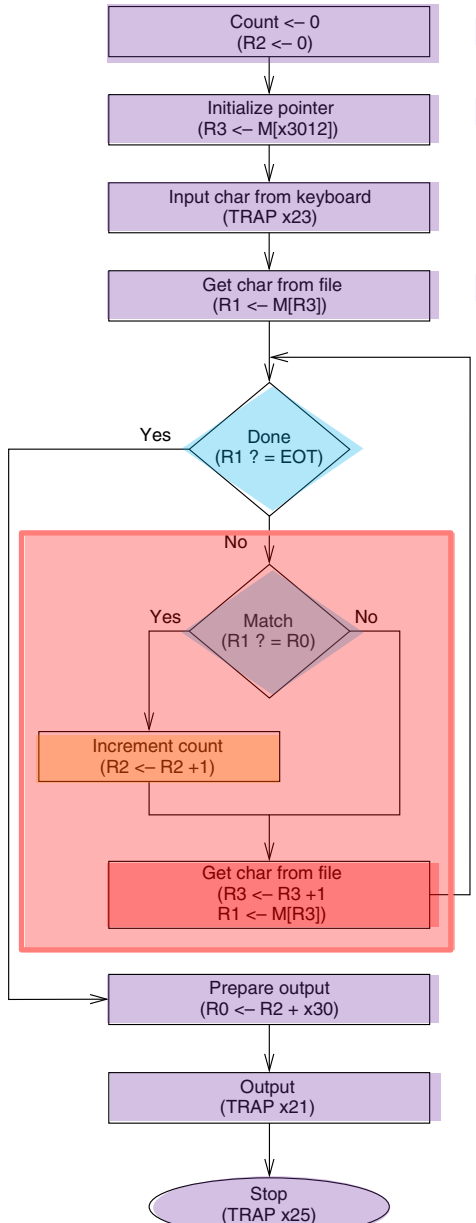
Constructs in an Example Program

- Let us see how to use the programming constructs in an example program
- The example program counts the number of occurrences of a character in a text file
- It uses sequential, conditional, and iterative constructs
- We will see how to write conditional and iterative constructs with conditional branches

Counting Occurrences of a Character

- We want to write a program that counts the occurrences of a character in a file
  - Get character-to-search from the keyboard (TRAP instr.)
  - The file finishes with the character EOT (End Of Text)
    - That is called a sentinel
    - In this example, EOT = 4
  - Output result to the monitor (TRAP instr.)





R2: counter

R3: initial address

Input char

Read char from file

Check if end of file

Is it the searched char?

Increment R2

Increment address
Read char from file

Move output to R0

Output counter

Halt the program

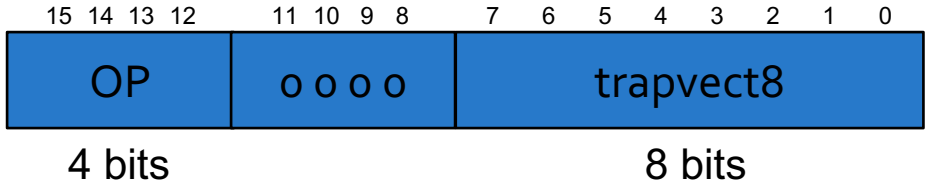
TRAP Instruction

- TRAP invokes an OS service call

LC-3 assembly

```
TRAP 0x23;
```

Machine Code



- OP = 1111
- trapvect8 = service call
  - 0x23 = Input a character from the keyboard
  - 0x21 = Output a character to the monitor
  - 0x25 = Halt the program

Counting Occurrences of a Char in LC-3

- We use conditional branch instructions to create loops and if statements

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize counter
x3001	LD	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 = M[0x3012] // initial address
x3002	TRAP	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	TRAP 0x23 // input char to R0
x3003	LDR	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 = M[R3] // char from file
x3004	ADD	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 = R1 – 4 // char – EOT
x3005	BR	0	0	0	0	z	0	0	0	0	0	0	1	0	0	0	BRz 0x300E // check if end of file
x3006	NOT	1	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 = NOT(R1) // subtract char from file from input char for comparison
x3007	ADD	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 = R1 + 1
x3008	ADD	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 = R1 + R0
x3009	BR	0	0	0	n	0	p	0	0	0	0	0	0	0	0	1	BRnp 0x300B
x300A	ADD	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 = R2 + 1 // increment the counter
x300B	ADD	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 = R3 + 1 // increment address
x300C	LDR	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 = M[R3] // char from file
x300D	BR	0	0	0	n	z	p	1	1	1	1	1	0	1	1	0	BRnzp 0x3004
x300E	LD	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 = M[0x3013] // output counter to monitor with TRAP
x300F	ADD	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0 = R0 + R2
x3010	TRAP	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	TRAP 0x21
x3011	AND	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP 0x25
x3012	Starting address of file																
x3013	ASCII TEMPLATE																

Programming Constructs in LC-3

- Let us do some reverse engineering to identify conditional constructs and iterative constructs

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	AND				0	1	0	0	1	0	1	0	0	0	0	0
x3001	LD				0	1	1	0	0	0	0	1	0	0	0	0
x3002	TRAP				0	0	0	0	0	0	1	0	0	0	1	1
x3003	LDR				0	0	1	0	1	1	0	0	0	0	0	0
x3004	ADD				1	0	0	0	0	1	1	1	1	1	0	0
x3005	BR				0	z	0	0	0	0	0	0	1	0	0	0
x3006	NOT				0	0	1	0	0	1	1	1	1	1	1	1
x3007	ADD				0	0	1	0	0	1	1	0	0	0	0	1
x3008	ADD				0	0	1	0	0	1	0	0	0	0	0	0
x3009	BR				n	0	p	0	0	0	0	0	0	0	0	1
x300A	ADD				0	1	0	0	1	0	1	0	0	0	0	1
x300B	ADD				0	1	1	0	1	1	1	0	0	0	0	1
x300C	LDR				0	0	1	0	1	1	0	0	0	0	0	0
x300D	BR				n	z	p	1	1	1	1	1	0	1	1	0
x300E	LD				0	0	0	0	0	0	0	0	0	1	0	0
x300F	ADD				0	0	0	0	0	0	0	0	0	0	1	0
x3010	TRAP				0	0	0	0	0	0	1	0	0	0	0	1
x3011	AND				0	0	0	0	0	0	1	0	0	1	0	1
x3012	Starting address of file															
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

```
while (R1 != EOT) {
...
}
```

```
R4 = R1 – 4 // char – EOT
BRz 0x300E // check if end of file
R1 = NOT(R1)   // subtract char from
R1 = R1 + 1    file from input char
R1 = R1 + R0   for comparison
BRnp 0x300B
R2 = R2 + 1 // increment the counter
BRnzp 0x3004
```

```
if (R1 == R0) {
... // increment the counter
}
```

Hardware Design

Assembly Programming: Debugging

```
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
```

Debugging

- Debugging is the process of removing errors in programs
- It consists of tracing the program, i.e., keeping track of the sequence of instructions that have been executed and the results produced by each instruction
- A useful technique is to partition the program into parts, often referred to as modules, and examine the results computed in each module
- High-level language (e.g., C programming language) debuggers: dbx, gdb, Visual Studio debugger
- Machine code debugging: Elementary interactive debugging operations

Interactive Debugging

- When debugging interactively, it is important to be able to
  - 1. Deposit values in memory and in registers, in order to test the execution of a part of a program in isolation
  - 2. Execute instruction sequences in a program by using
    - RUN command: execute until HALT instruction or a breakpoint
    - STEP N command: execute a fixed number (N) of instructions
  - 3. Stop execution when desired
    - SET BREAKPOINT command: stop execution at a specific instruction in a program
  - 4. Examine what is in memory and registers at any point in the program

Example: Multiplying in LC-3 (Buggy)

- A program is necessary to multiply, since LC-3 does not have multiply instruction
  - The following program multiplies R4 and R5
  - Initially, R4 = 10 and R5 = 3
  - The program produces 40. What went wrong?
  - It is useful to annotate each instruction

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 – 1
x3203	0	0	0	0	0	z	p	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

Debugging the Multiply Program

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	ADD	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	ADD	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 – 1
x3203	0	BR	0	0	0	0	z	p	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	HALT	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

?

- We examine the contents of all registers after the execution of each instruction

PC	R2	R4	R5	
x3201	0	10	3	
x3202	10	10	3	
x3203	10	10	2	
x3201	10	10	2	
x3202	20	10	2	
x3203	20	10	1	
x3201	20	10	1	
x3202	30	10	1	← Correct result
x3203	30	10	0	← BR should not be taken if R5 = 0
x3201	30	10	0	
x3202	40	10	0	
x3203	40	10	−1	
x3204	40	10	−1	
	40	10	−1	

The branch condition codes were set wrong. The conditional branch should only be taken if R5 is positive

Correct instruction:

```
BRp  #-3      // BRp 0x3201
```

Debugging the Multiply Program

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	ADD	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	ADD	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 – 1
x3203	0	BR	0	0	0	0	z	p	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	HALT	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

?

- We could use a breakpoint to save some work
- Setting a breakpoint in 0x3203 (BR) allows us to examine the results of each iteration of the loop

PC	R2	R4	R5	
x3203	10	10	2	
x3203	20	10	1	
x3203	30	10	0	← BR should not be taken if R5 = 0
x3203	40	10	−1	

One last question:
Does this program work if the initial value of R5 is 0?

A good test should also consider the corner cases,
i.e., unusual values that the programmer might fail to consider

Hardware Design

Assembly Programming:
Conditional Statements and Loops in MIPS Assembly

```
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
```

If Statement

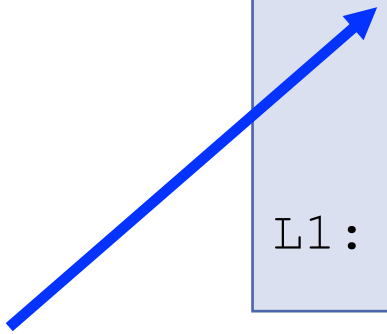
- In MIPS, we create conditional constructs with conditional branches (e.g., beq, bne…)

High-level code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS assembly

```
# $s0 = f, $s1 = g
# $s2 = h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```



Branch not equal: Compares two values (`$s3=i, $s4=j`) and jumps if they are different

If-Else Statement

- We use the unconditional branch (i.e., j) to skip the "else" subtask if the "if" subtask is the correct one

High-level code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

1. Compare two values ($s3=i, $s4=j) and, if they are different, jump to L1, to execute the "else" subtask

2. Jump to done, after executing the "if" subtask

MIPS assembly

```
# $s0 = f, $s1 = g,
# $s2 = h
# $s3 = i, $s4 = j

      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

While Loop

- As in LC-3, the conditional branch (i.e., beq) checks the condition, and the unconditional branch (i.e., j) jumps to the beginning of the loop

High-level code

```
// determines the power
// of 2 equal to 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

1. Conditional branch to check if the condition still holds

2. Unconditional branch to the beginning of the loop

MIPS assembly

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while

done:
```

For Loop

- The implementation of the "for" loop is similar to the "while" loop

High-level code

```
// add the numbers from 0 to 9

int sum = 0;
int i;
for (i = 0; i != 10; i = i+1)
{
  sum = sum + i;
}
```

1. Conditional branch to check if the condition still holds

2. Unconditional branch to the beginning of the loop

MIPS assembly

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

For Loop Using SLT

- We use slt (i.e., set less than) for the "less than" comparison

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i*2)
{
  sum = sum + i;
}
```

Set less than

```
$t1 = $s0 < $t0 ? 1:0
```

Shift left logical

MIPS assembly

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

Initialize sum and i

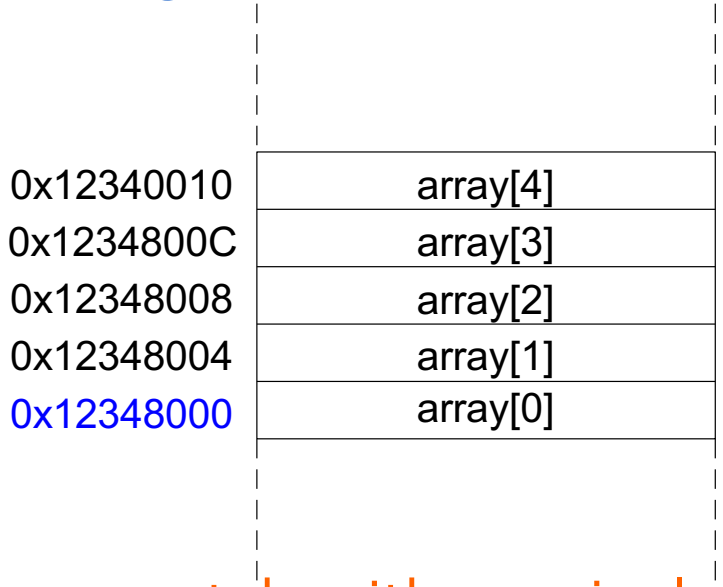
Hardware Design

Assembly Programming: Arrays in MIPS

```
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
```

Arrays

- Accessing an array requires loading the base address into a register



- In MIPS, this is something we cannot do with one single immediate operation
- Load upper immediate + OR immediate

```
lui   $s0, 0x1234
ori   $s0, $s0, 0x8000
```

Arrays: Code Example

- We first load the base address of the array into a register (e.g., $so) using lui and ori

High-level code

```
int array[5];

array[0] = array[0] * 2;

array[1] = array[1] * 2;
```

MIPS assembly

```
# array base address = $s0
# Initialize $s0 to 0x12348000
lui   $s0, 0x1234
ori   $s0, $s0, 0x8000

lw    $t1, 0($s0)
sll   $t1, $t1, 1
sw    $t1, 0($s0)
lw    $t1, 4($s0)
sll   $t1, $t1, 1
sw    $t1, 4($s0)
```

Hardware Design

Assembly Programming: Function Calls

```
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
```

Function Calls

- Why functions (i.e., procedures)?
  - Frequently accessed code
  - Make a program more modular and readable
- Functions have arguments and return value
- Caller: calling function
  - main()
- Callee: called function
  - sum()

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Calls: Conventions

- Conventions
  - Caller
    - passes arguments
      - jumps to callee
  - Callee
    - performs the procedure
    - returns the result to caller
    - returns to the point of call
    - must not overwrite registers or memory needed by the caller

Function Calls in MIPS and LC-3

- Conventions in MIPS and LC-3
  - Call procedure
    - MIPS: Jump and link (jal)
    - LC-3: Jump to Subroutine (JSR, JSRR)
  - Return from procedure
    - MIPS: Jump register (jr)
    - LC-3: Return from Subroutine (RET)
  - Argument values
    - MIPS: $a0 - $a3
  - Return value
    - MIPS: $v0

Function Calls: Simple Example

High-level code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly

```
0x00400200 main: jal simple
0x00400204       add $s0,$s1,$s2



...
0x00401020 simple: jr $ra
```

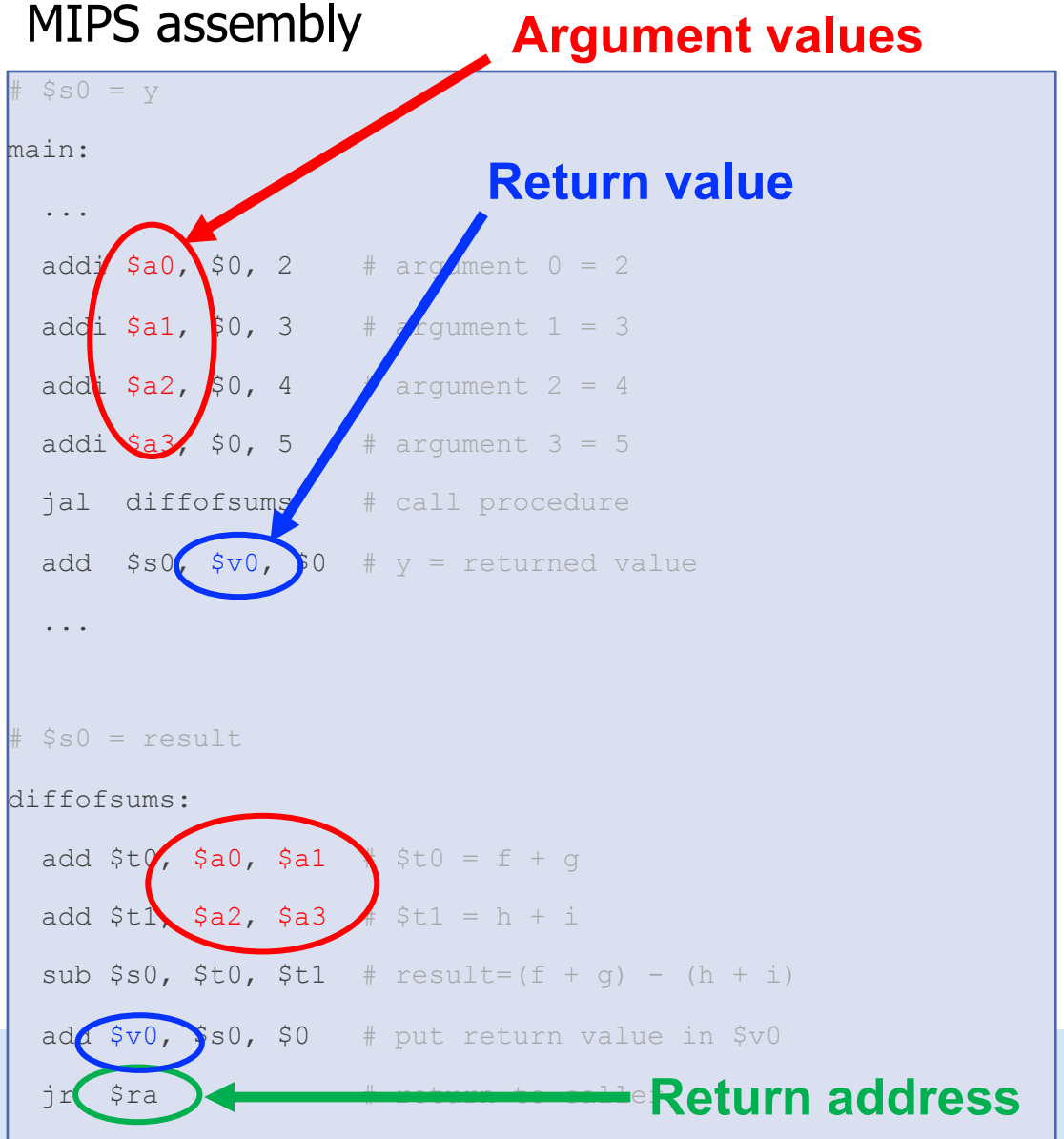
- jal jumps to simple() and saves PC+4 in the return address register ($ra)
  - $ra = 0x00400204
  - In LC-3, JSR(R) put the return address in R7
- jr $ra jumps to address in $ra (LC-3 uses RET instruction)

Function Calls: Code Example

High-level code

```
int main()
{
  int y;
  ...
  // 4 arguments
  y = diffofsums(2, 3, 4, 5);
  ...
}

int diffofsums(int f, int g,
  int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  // return value
  return result;
}
```

MIPS assembly



Function Calls: Need for the Stack

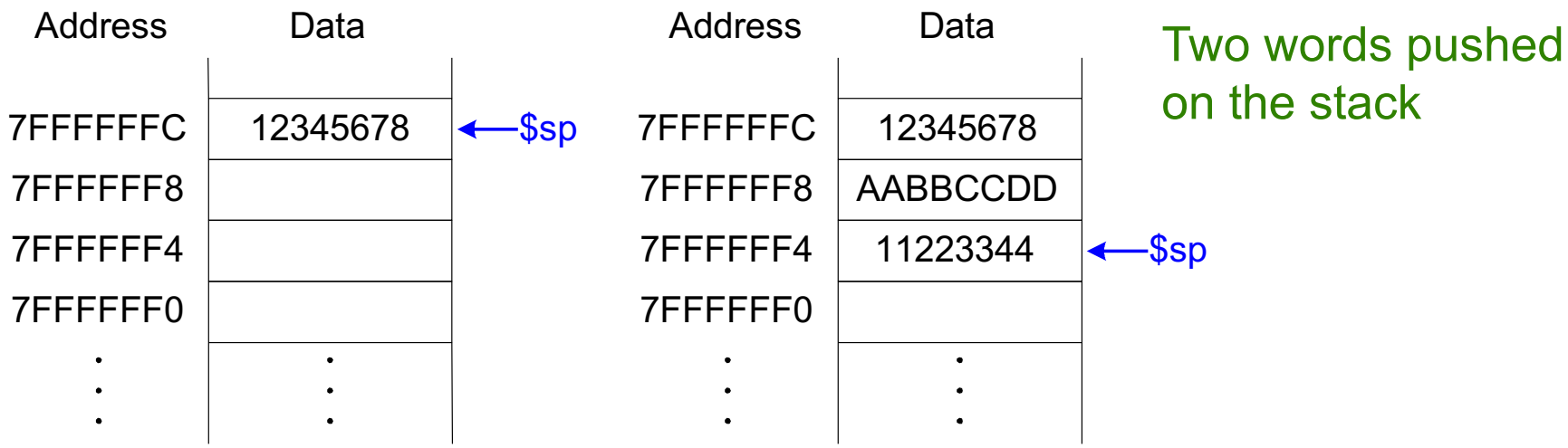
MIPS assembly

```
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result=(f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

- What if the main function was using some of those registers?
  - `$t0`, `$t1`, `$s0`
- They could be overwritten by the function
- We can use the stack to temporarily store registers

The Stack

- The stack is a memory area used to save local variables
- It is a Last-In-First-Out (LIFO) queue
- The stack pointer ($sp) points to the top of the stack
  - It grows down in MIPS



The Stack: MIPS assembly Code Example

```
diffofsums:
   addi $sp, $sp, -12   # allocate space on stack to store 3
                        registers
   sw   $s0, 8($sp)     # save $s0 on stack
   sw   $t0, 4($sp)     # save $t0 on stack
   sw   $t1, 0($sp)     # save $t1 on stack
   add $t0, $a0, $a1    # $t0 = f + g
   add $t1, $a2, $a3    # $t1 = h + i
   sub $s0, $t0, $t1    # result=(f + g) - (h + i)
   add $v0, $s0, $0     # put return value in $v0
   lw   $t1, 0($sp)     # restore $t1 from stack
   lw   $t0, 4($sp)     # restore $t0 from stack
   lw   $s0, 8($sp)     # restore $s0 from stack
   addi $sp, $sp, 12    # deallocate stack space
  jr  $ra               # return to caller
```

- Saving and restoring all registers requires a lot of effort
- In MIPS, there is a convention about temporary registers (i.e., $to-$t9): There is no need to save them
  - Programmers can use them for temporary/partial results

MIPS Stack: Register Saving Convention

```
diffofsums:
   addi $sp, $sp, -4    # allocate space on stack to store 1 register
   sw   $s0, 0($sp)     # save $s0 on stack


   add $t0, $a0, $a1    # $t0 = f + g
   add $t1, $a2, $a3    # $t1 = h + i
   sub $s0, $t0, $t1    # result=(f + g) - (h + i)
   add $v0, $s0, $0     # put return value in $v0


   lw   $s0, 0($sp)     # restore $s0 from stack
   addi $sp, $sp, 4     # deallocate stack space
   jr  $ra              # return to caller
```

- Temporary registers $to-$t9 are non-preserved registers. They are not saved, thus, they can be overwritten by the function
- Registers $so-$s7 are preserved (saved; callee-saved) registers

KING'S College LONDON

Hardware Design

Lecture 3: ISA (III), Microarchitecture, and Assembly

Dr. Haiyu Mao

05.02.2026