1. Instruction Set Architecture & Compiler Basics

2025 Fall
Hunjun Lee
hunjunlee@hanyang.ac.kr

Computing Problem Hierarchy

Algorithm
Programming
System Software
SW/HW Interface
Microarchitecture
Logic
Devices

OK, very basics first.

“How does your program run on computer?”

Basic components of a computer

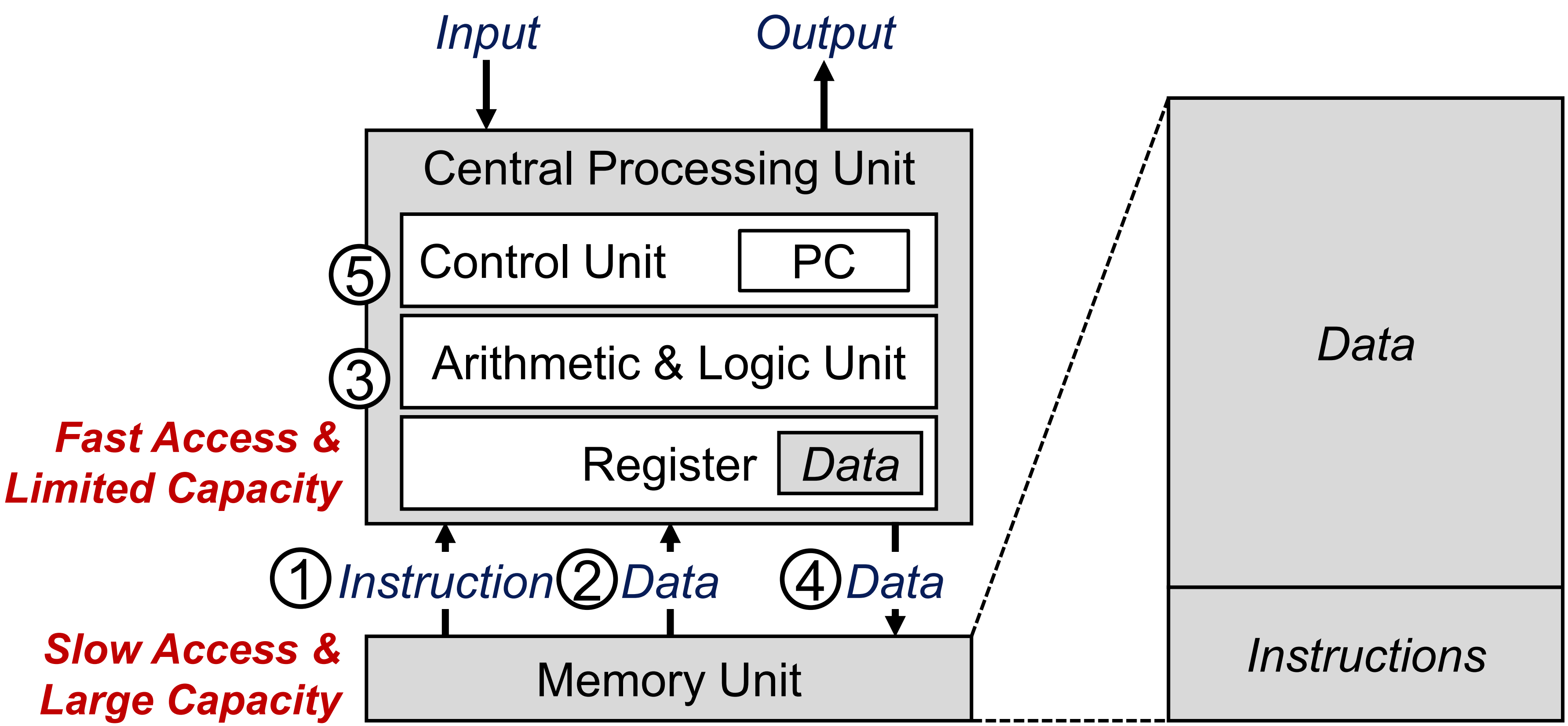
- To get a task done by a general-purpose computer, we need
  - A computer program
    - Specifies what computers must do!
  - The computer itself
    - To carry out the specified task

- Program: A sequence of instructions
  - Instruction: the smallest piece of specified work that the computer can carry out

- Instruction set: All possible instructions that a computer is designed to be able to carry out

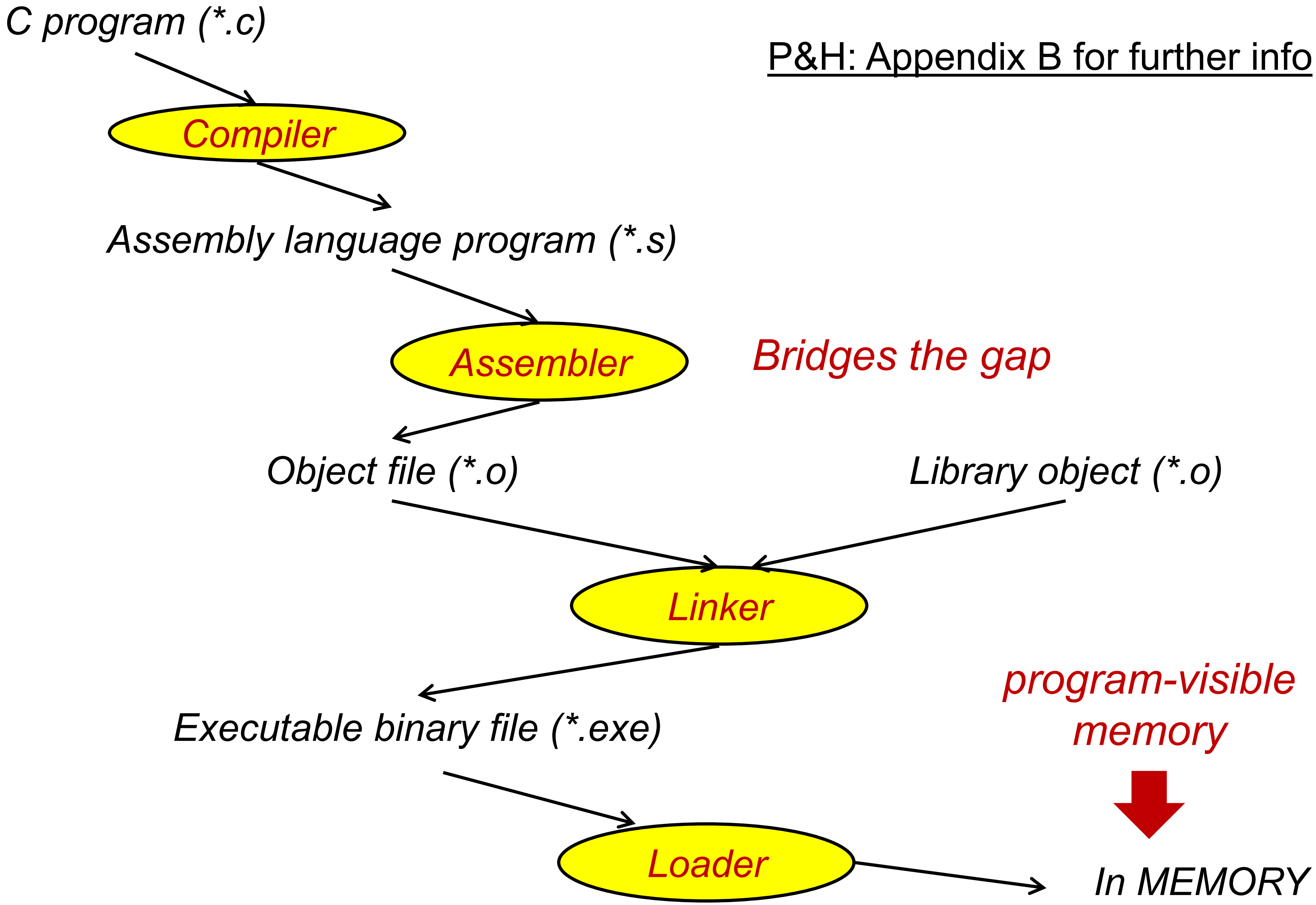
Von Neumann Architecture

- Both instructions and data are stored in the memory
- Instructions dictate (1) which and how data are manipulated and (2) which instruction should be next



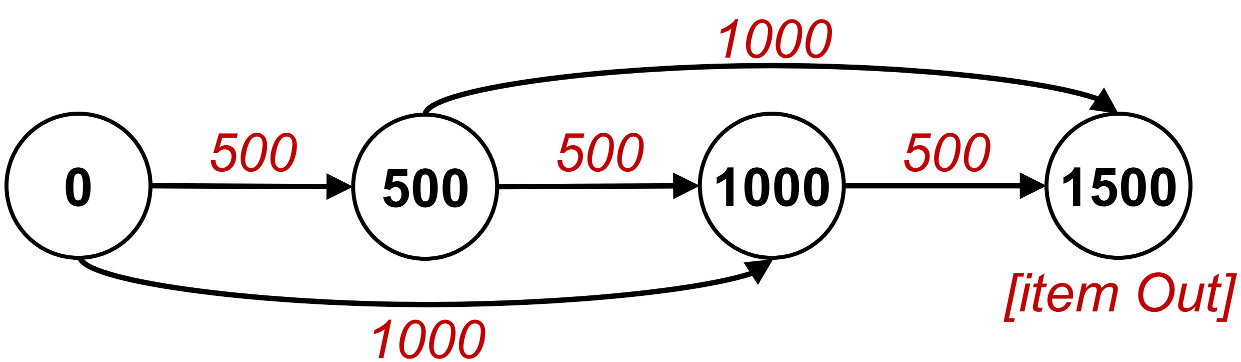
How to load a program to your computer?

P&H: Appendix B for further info

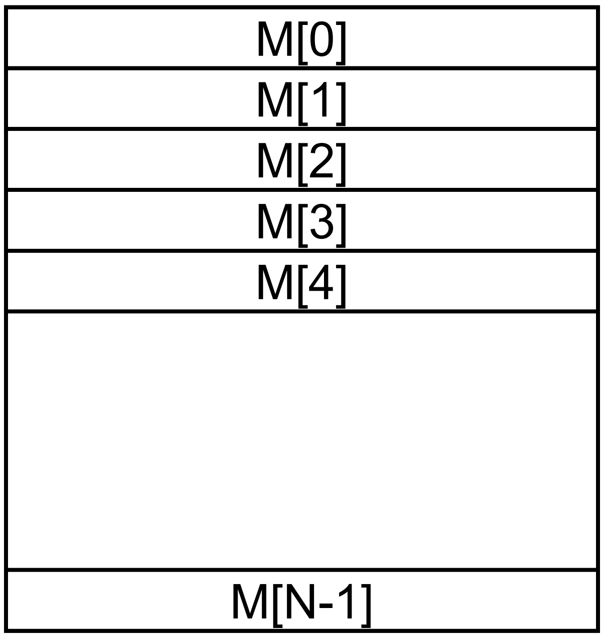


Recall: State machine

- Computers are essentially a complex state machine
- State machine (Ex. vending machine)
  - Condition: When a user inputs a total of 1500 ₩, the machine outputs a Coke
  - Input type: a user can input 500 ₩ coin and 1000 ₩ bill)
  - States: an amount of cash a user has inputted

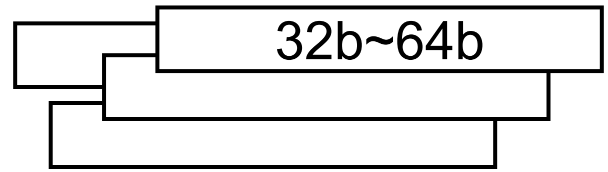


Programmer visible state
(a.k.a. architectural state)



Memory

Array of storage locations indexed by an address



Registers

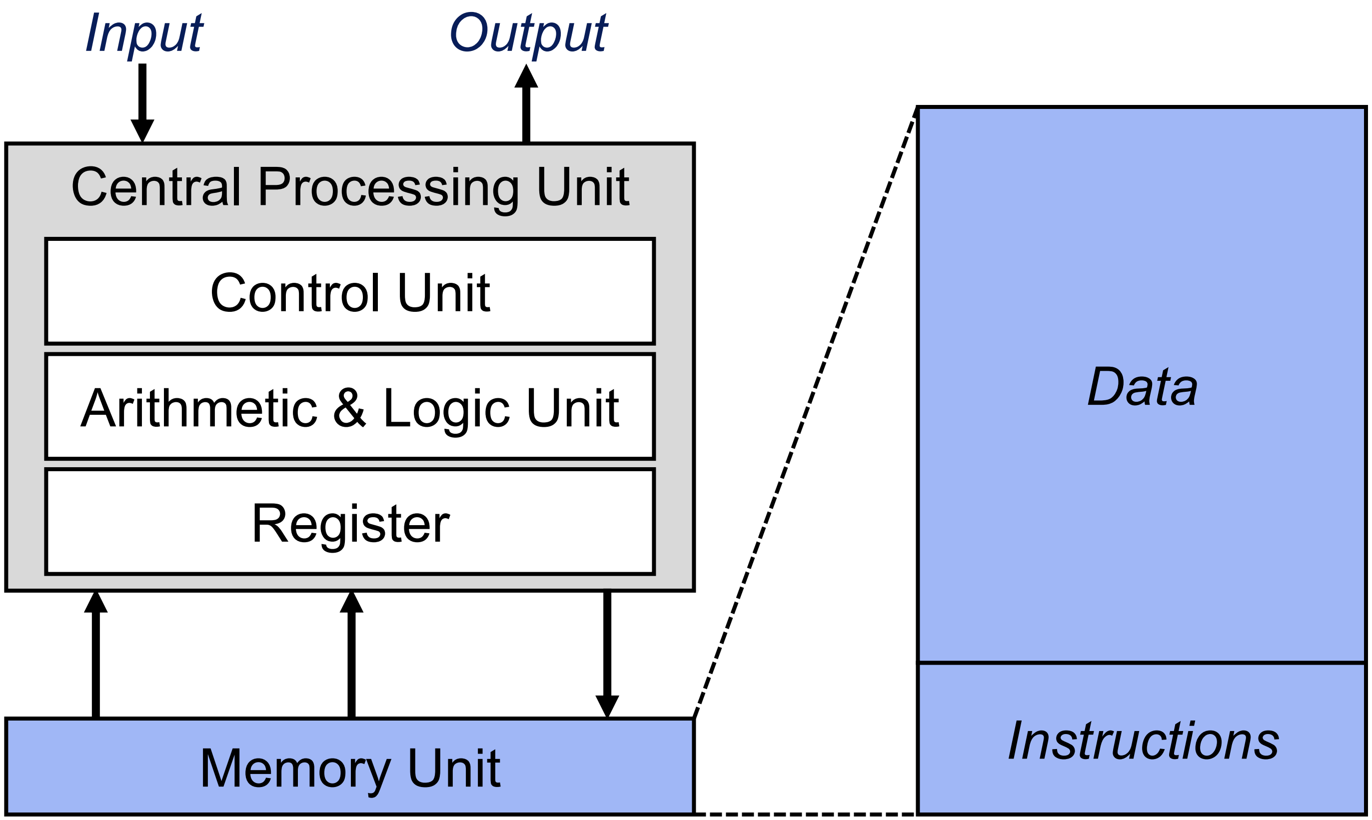
- Given special names in the ISA (as opposed to addresses)
- General vs. special purpose

Program Counter (32b~64b)

Memory address of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

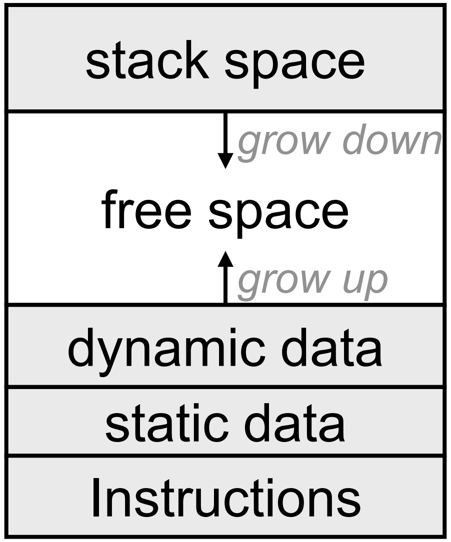
Von Neumann Architecture



Runtime storage organization

- The memory stores the following:
  - (1) program (instructions) and (2) data required in program execution
- Memory contains bits
  - Logically grouped into bytes (8 bits) and words (e.g., 8, 16, 32 bits)
  - The word size determines the instruction width, register size, …
- Address space: Total number of uniquely identifiable locations in memory (differs depending on the architecture)
  - In LC-3, the address space is 2^{16} (16-bit addresses)
  - In MIPS, the address space is 2^{32} (32-bit addresses)
  - In x86-64, the address space is (up to) 2^{48} (48-bit addresses)

There are four different parts in the storage!



How to maintain data for the program? – using stack

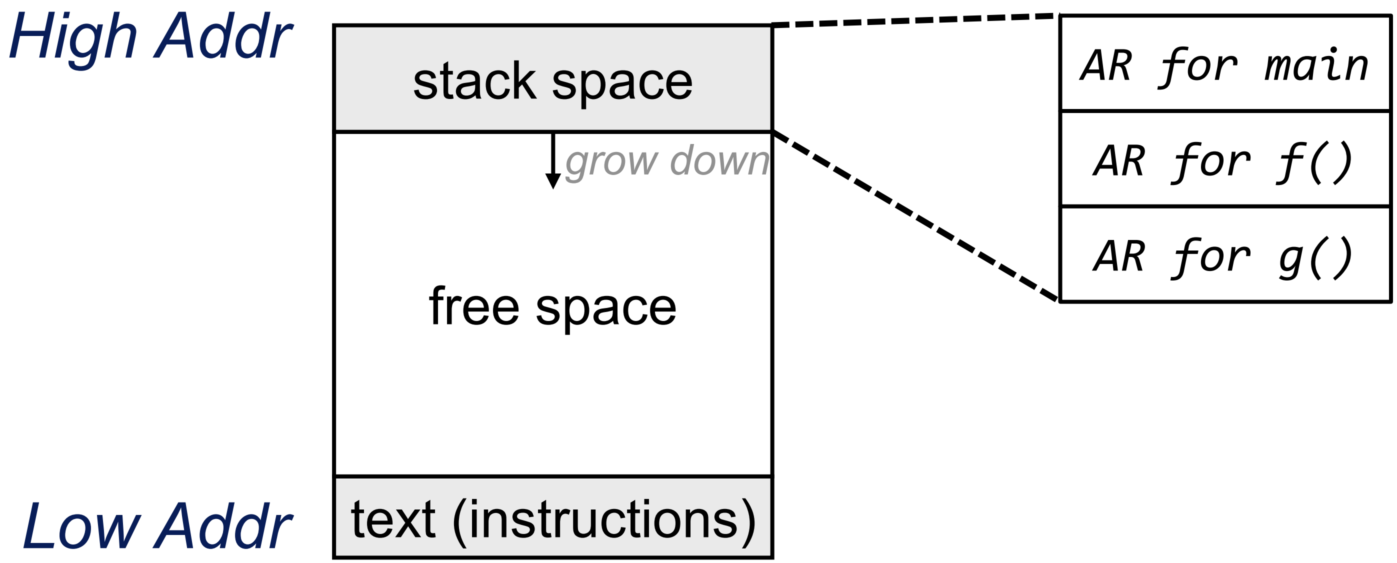
- We can utilize a “stack” to manage data for the active procedures (or functions)
- When P calls Q, then Q returns before P returns
  - Lifetimes of procedure activations (required data for the procedure) are properly nested
  - The activation depends on run-time behavior

```
int g() { return 1; }
int f() { return g(); }
void main() {
        g();
        f();
}
```

Step1	Step2	Step3	Step4	Step5	Step6	Step7
main	main	main	main	main	main	main
	g()		f()	f()	f()	
				g()		

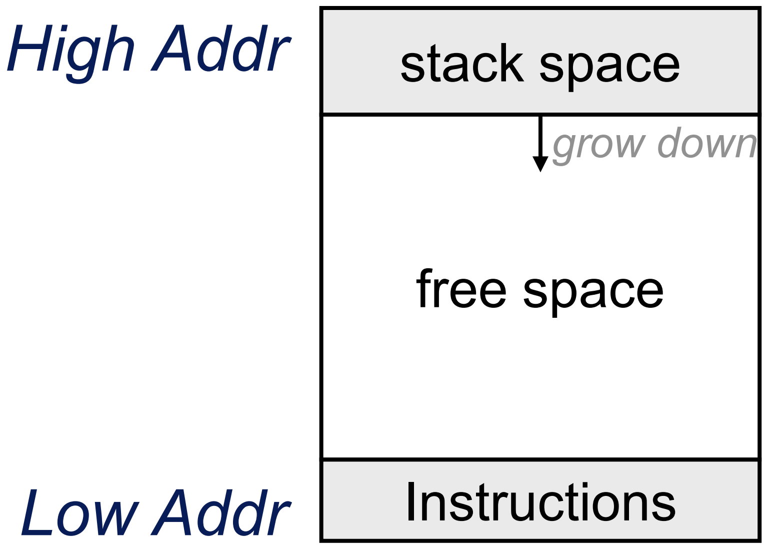
Stack management

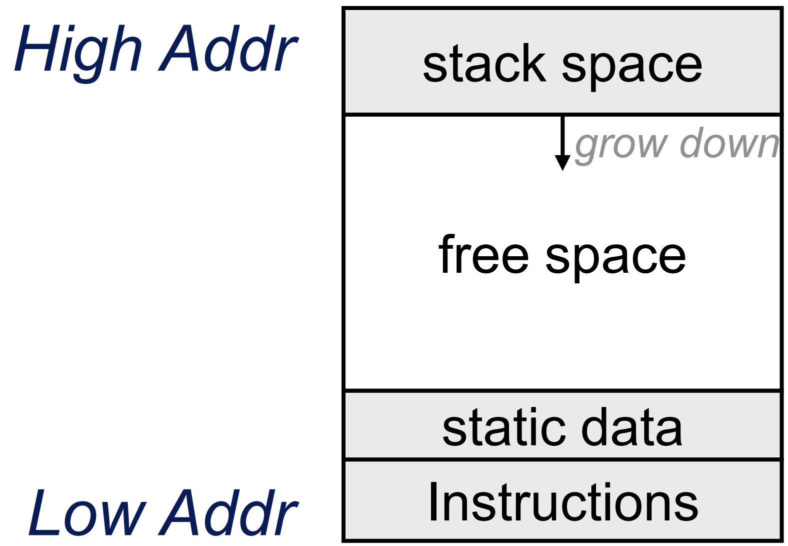
- Stack data is stored starting from the low address, which grows downwards
- The information to manage one function call is called activation record (AR) or frame



Global variables

- All references to a global variable point to the same object
  - It would be impossible (or inefficient) to store a global activation in an activation record
- Global variables are assigned a fixed address once (statically allocated)

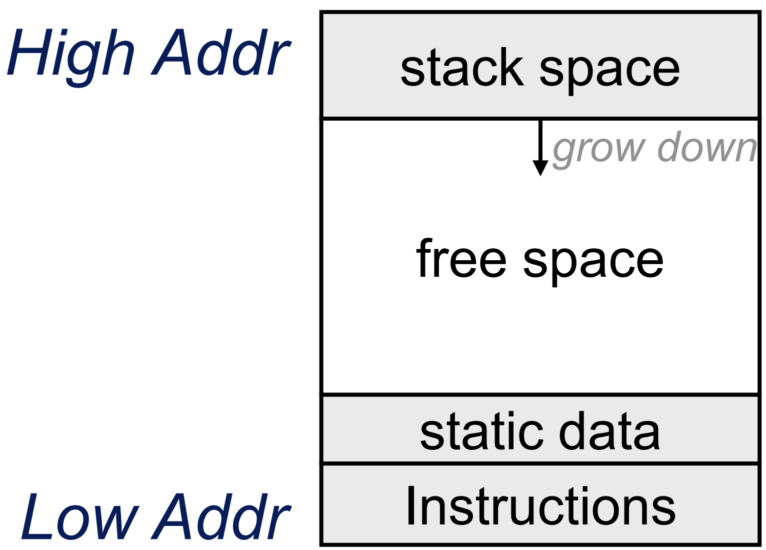


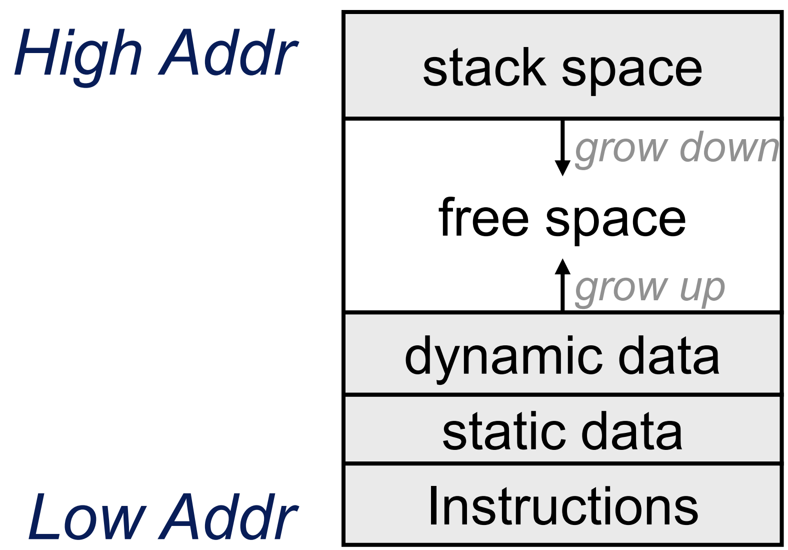


Dynamic allocation

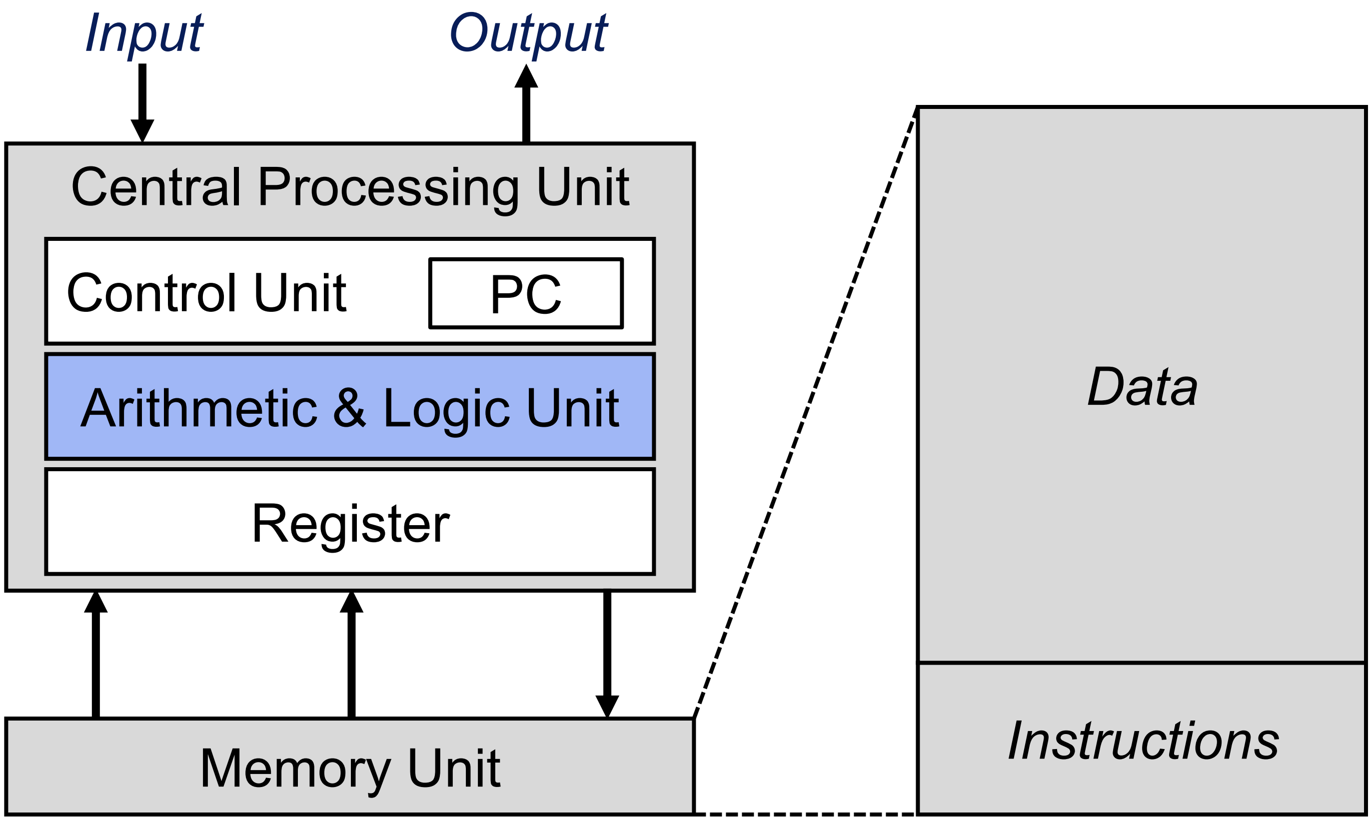
- The dynamically allocated value outlives the procedure that creates it (unless deleted beforehand)

- We rely on heap to store the dynamically allocated data





Von Neumann Architecture



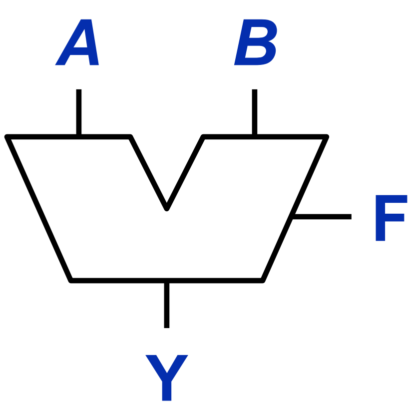
Processing units

- Performs the actual computation(s)
- The processing unit can consist of many functional units
- We start with a simple Arithmetic and Logic Unit (ALU), which executes computation and logic operations
  - MIPS: add, sub, mult, and, nor, …
- The ALU processes quantities that are referred to as words
  - Word length in MIPS is 32 bits

ALU (Arithmetic logical unit)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)

- Usually denoted with this symbol:

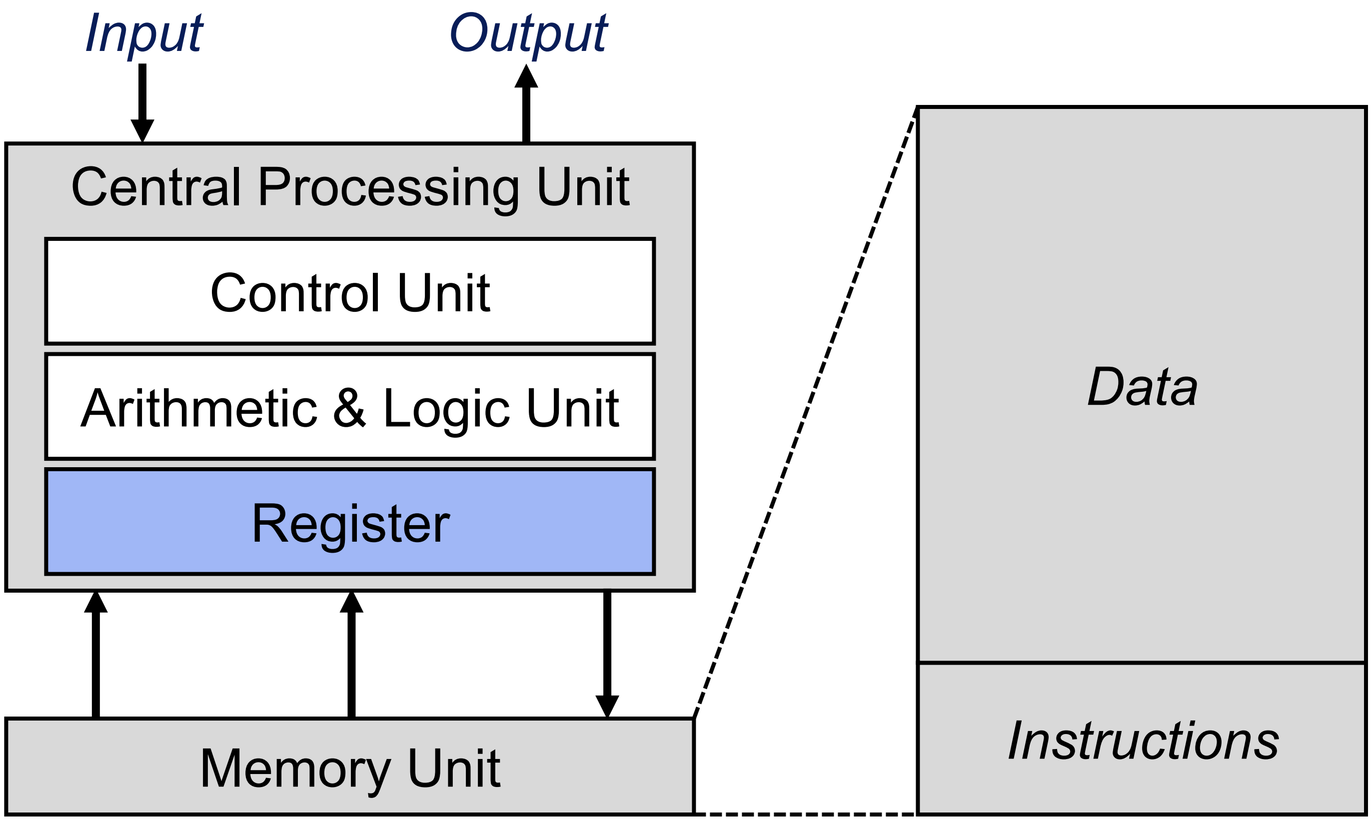


F[2:0]	Function
000	Y = A and B
001	Y = A or B
010	A + B
011	Not used
100	A – B
101	A * B
110	A / B
111	SLT

One of the examples

Von Neumann Architecture

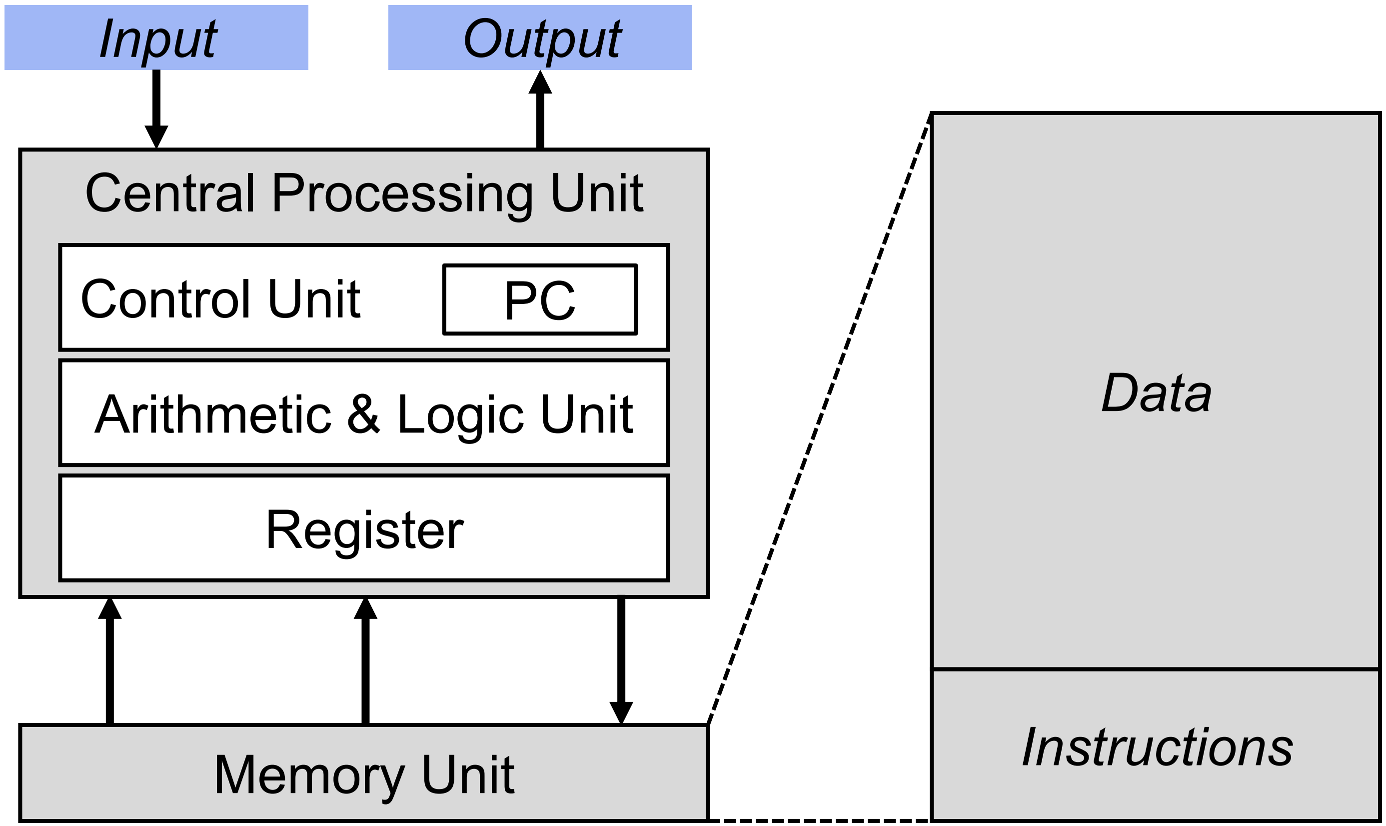
- Both instructions and data are stored in the memory
- Each instruction dictates (1) which and how data are manipulated and (2) which instruction should be next



Registers: fast storage

- **Memory** is large but slow while **registers** are fast and small
- Processing unit utilizes registers during operation
  - Ensure fast access to values to be processed in the ALU (stores temporary values)
  - Combinational read & synchronous write ➔ can execute read + ALU + write in a single cycle
- Register Set or Register File
  - Set of registers that can be manipulated by instructions
  - MIPS has 32 registers
    - R0 to R31: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits
  - There are some special-purpose registers (e.g., `$fp, $sp, …`)

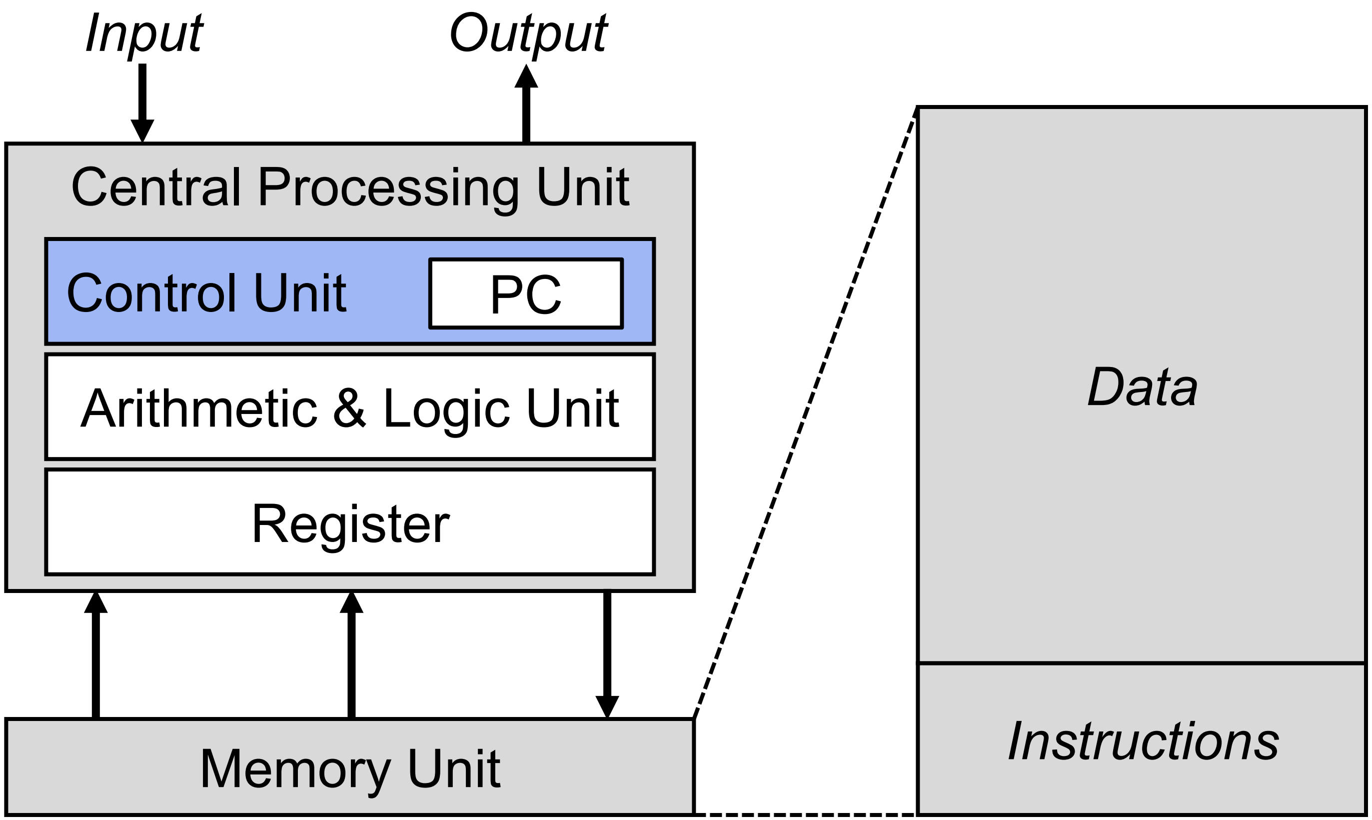
Von Neumann Architecture



Input and output

- Enable information to get into and out of a computer
- There are many input and output devices
- Input examples
  - keyboard, mouse, scanner, disks, …
- Output examples
  - monitor, printer, disks, …

Von Neumann Architecture



Control unit

- Enables a step-by-step execution of a program
  - Proceeds through each instruction in a program in sequence

- Keeps track of which instruction is being processed, via an instruction register

- Keeps track of which instruction to process next, using a program counter (PC) and determines which instruction to process next

Wrap up

- **Memory Unit:** stores (1) program (instructions) and (2) data required in program execution

- **Arithmetic & Logical Unit:** performs the actual computation(s)

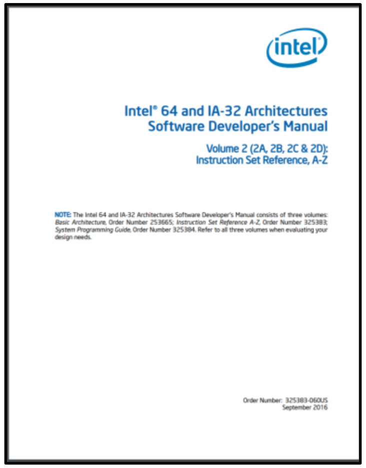
- **Input & Output:** enables information to get in and out of the computer

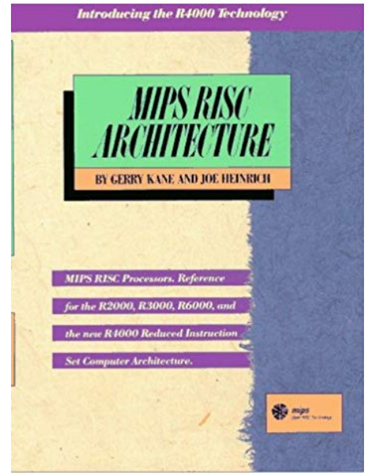
- **Control Unit:** enables a step-by-step execution of a program

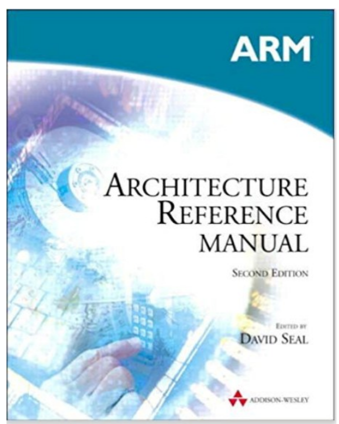
Instruction Set Architecture (ISA)

“User’s manual for the computer”

Architecture Manuals







Each vendor specifies the instruction sets in a different way

Architecture*

- “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 data flow and controls, the logical design, and the physical implementation.”

--- footnote on page 1, Architecture of the IBM System/360,
Amdahl, Blaauw and Brooks, 1964.

How to specify what a computer does?

- **Architecture** Level
  - Car: driving manual & operation manual

    // you don't have to be a car mechanic to drive a car.
  - Computer : program manual

    // you don't have to be a circuit designer to program a computer.

- **Microarchitecture** (implementation) Level
  - A particular car design has a certain configuration of electrical/mechanical components (e.g., v8 engine vs. v4 engine)
  - A particular computer design has a certain configuration of datapath and control logic units (e.g., adder type, cache, …)

What are specified/decided in an ISA?

- Data format and size
  - character, binary, decimal, floating point, …
- "Programmer Visible State" **(a.k.a. architectural state)**
  - memory, registers, program counter (PC), etc.
- Instructions: how to **"change"** the programmer visible state?
  - What to perform and what to perform next
  - Where the operands are
- How to interface with the outside world?
- Protection and privileged operations
- Software conventions

Often, you compromise performance for future scalability and compatibility

General Instruction Classes

- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  1) Load operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  4) Update PC to the next sequential instruction
- **Data movement operations** (e.g., load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  3) Update PC to the next sequential instruction
- **Control flow operations** (e.g., branch, jump)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC $\leftarrow$ target address
     else PC $\leftarrow$ next seq. instruction

Generally defined to be atomic

Instructions for operations

- Different ISAs utilize different instruction for operations
- Number of Operands
  - **Monadic** `OP in2`
  - **Binatic** `OP inout, in2`
    - Save memory (smaller instruction size)
  - **Triadic** `OP out, in1, in2`

- Can ALU operands be in memory?
  - **No!** `ADD r1 r2 r3 (r1~r3 are registers)`
    - You should load memory to the register before ALU operations
  - **Yes!** `ADD r2, r1, [2000]`

Different methods depending on the ISA

Instructions for memory addressing

- Absolute `lw rt, 10000`
  - `lw`: load word use immediate value as address
- Register Indirect $\texttt{lw rt, (r}_{\texttt{base}}\texttt{)}$
  - use Register[r_{base}] as address
- Displaced or based $\texttt{lw rt, offset(r}_{\texttt{base}}\texttt{)}$
  - use offset+ Register[r_{base}] as address
- Indexed $\texttt{lw rt, (r}_{\texttt{base}}\texttt{, r}_{\texttt{index}}\texttt{)}$
  - use Register[r_{base}]+ Register[r_{index}] as address
- Memory Indirect $\texttt{lw rt ((r}_{\texttt{base}}\texttt{))}$
  - use value at M[Register[r_{base}]] as address

Complicated memory addressing modes simplify program

MIPS RISC

- Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - Only few alternatives exist to do the same thing
- Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - Memory can be accessed by only load and store instructions
    → **“Load-store architecture”**
- Simple branches
  - Limited varieties of branch conditions and targets
- Simple instruction encoding
  - All instructions encoded **in the same number of bits**
  - Only a few formats

Such ISA intended for compiler advances rather than assembly programmers

Evolution of ISA

- Why were the earlier ISAs so simple?
  - Technology limitation
  - Inexperience, lack of precedence
- Why did it get so complicated later?
  - Complex instruction set architecture (CISC)
  - Ease of assembly programming
  - Lack of memory size and performance (in 1970s~80s)
  - Micro-programmed implementation
- Why did it become simple again?
  - Reduced instruction set architecture (RISC)
  - Memory size and speed (cache!)
  - **Compilers**

What about x86 (Intel and AMD)?

RISC vs. CISC

- RISC (Reduced instruction set architecture)
  - The hardware exposes only basic operations as an ISA
  - Simpler instruction (+ simple decoding)
  - Fixed-width instruction (one word)

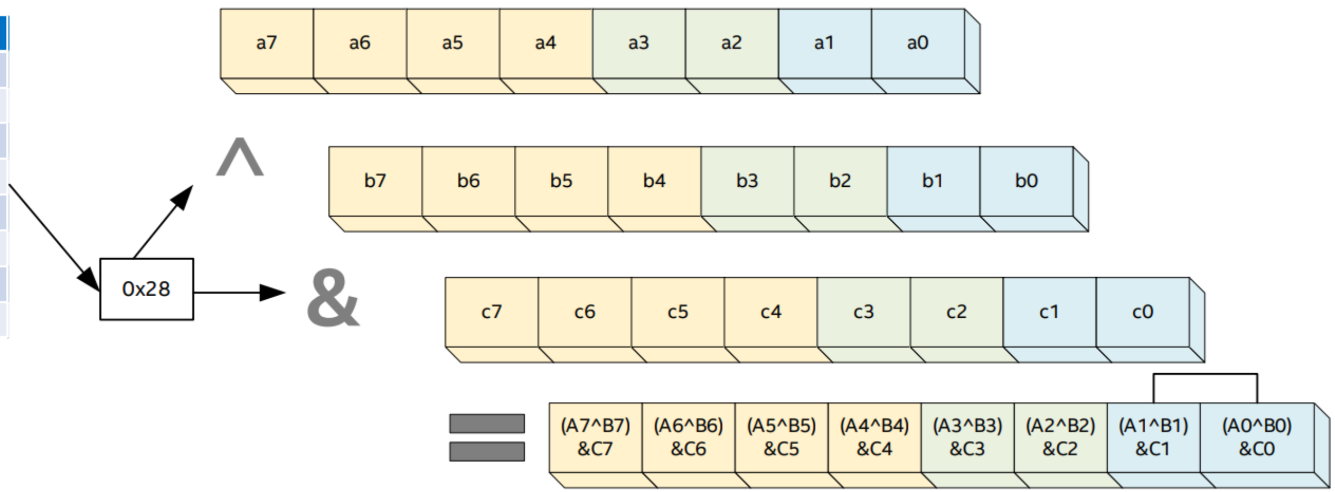
- CISC (Complex instruction set architecture)
  - Exposes more complex operations (combination of multiple RISC-style operations)
  - E.g., A single instruction may … load the data from both memory and register, perform addition, and write back the result
    - Remember `ADD r2, r1, [2000]`

Modern CPUs are RISC, but x86 (Intel + AMD) are CISC
Why??

ISA extension in modern processors - 1

- Intel AVX enables vector operations by defining an Intel AVX-512 ISA

Bit	A	B	C	(A ^ B) & C
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

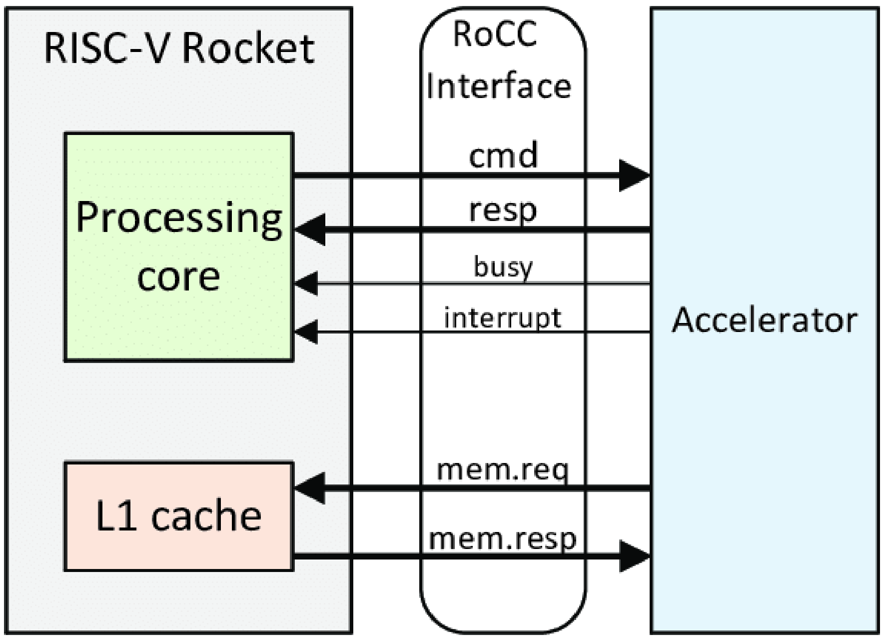


INSTRUCTION SET	C INTRINSIC FORM OF INSTRUCTION
AVX-512	__m512i _mm512_ternarylogic_epi64 (__m512i a, __m512i b, __m512i c, int imm8)
AVX-512	__m512i _mm512_mask_ternarylogic_epi64 (__m512i src, __mmask8 k, __m512i a, __m512i b, int imm8)
AVX-512	__m512i _mm512_maskz_ternarylogic_epi64 (__mmask8 k, __m512i a, __m512i b, __m512i c, int imm8)

ISA extension in modern processors - 2

- Rocket custom coprocessor (RoCC) interface in RISC-V
  - The computer architects can add a custom coprocessor
  - The RoCC interface controls the coprocessor

Controls the accelerator using RoCC



Add a custom accelerator (e.g., NPU)

```
#define ROCC_INSTRUCTION_0_R_R(x, rs1, rs2, func7){
      asm volatile(
      ".insn r " STR(CAT(CUSTOM_, x)) ", "
      STR(0x3) ", " STR(func7) ", x0, %0, %1”
      :
      : "r"(rs1), "r"(rs2));
 }
```

Wrap-up: Terminologies

- Instruction Set Architecture (ISA)
  - The machine behavior as observable and controllable by the programmer
- Instruction Set
  - The set of commands understood by the computer
- Assembly Code
  - A collection of instructions expressed in “textual” format

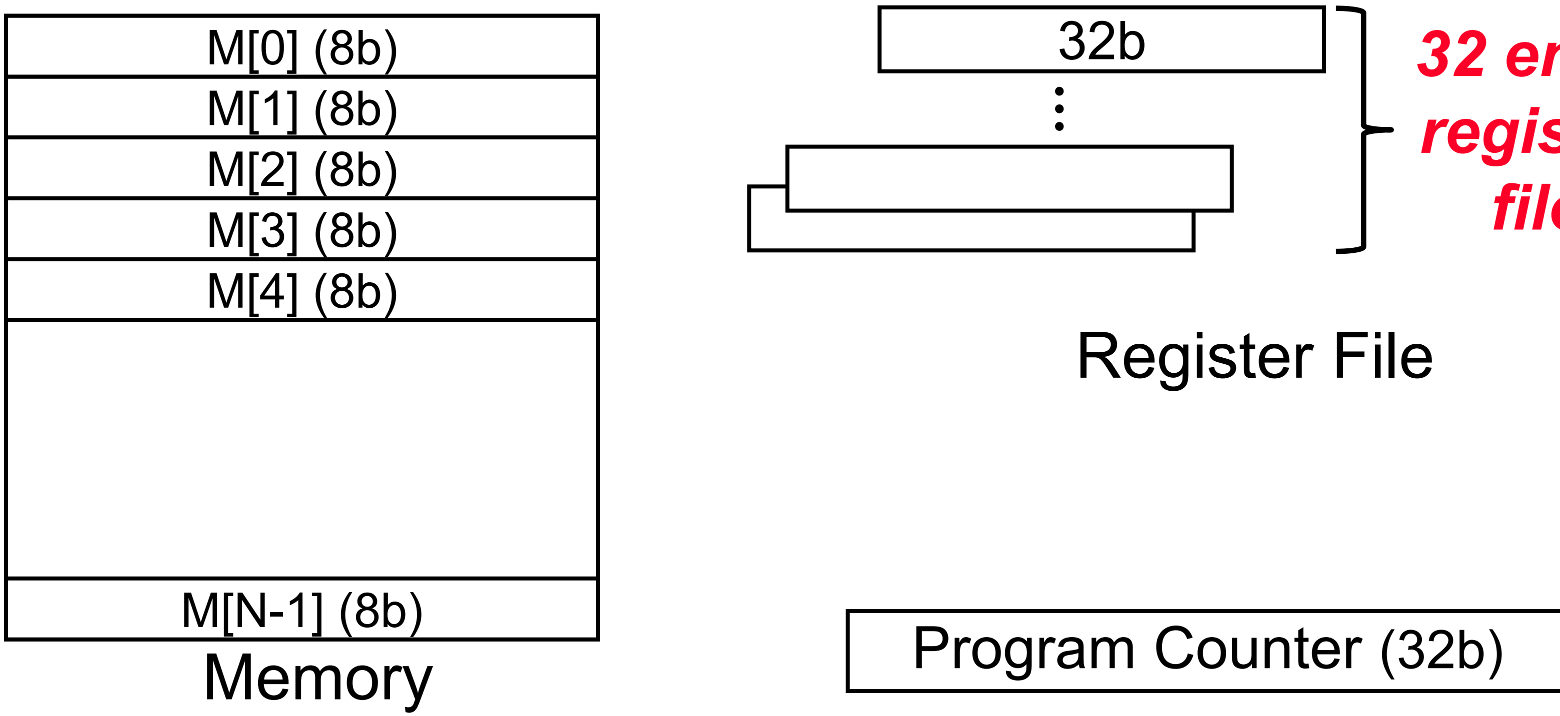
    e.g. Add r1, r2, r3
  - Converted to machine code by an assembler
  - One-to-one correspondence with machine code
- Machine Code
  - A collection of instructions encoded in binary format 0101000…
  - Directly consumable by the hardware

Let’s dive into a concrete example

MIPS ISA (32bit)

MIPS architectural state

Word addressable



32 entry register file

32bit address space + Byte addressable

32bit instruction

MIPS architecture has (1) 32-bit word size and (2) a 32-entry register file

MIPS instruction formats

- Three simple formats
  - R-type, 3 register operands

0	rs	rt	rd	shamt	funct	R-type
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit	

  - I-type, 2 register operands and 16-bit immediate operand

opcode	rs	rt	immediate	I-type
6-bit	5-bit	5-bit	16-bit	

  - J-type, 26-bit immediate operand

opcode	immediate	J-type
6-bit	26-bit	

- Simple Decoding
  - 4 bytes per instruction, regardless of format (fixed size)
  - Must be 4-byte aligned (2 LSB of PC must be 2b'00)
  - Format and fields readily extractable

R-type Instructions

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

`R-type`

- `funct:` type of ALU operation
  - Arithmetic: {signed, unsigned} x {ADD, SUB, MULT, DIV, …}
  - Logical: {AND, OR, XOR, NOR, …}
  - Shift: {Left, Right-Logical, Right-Arithmetic}
- `shamt:` shift amount (only used for shift operation)
- Assembly and semantics
  - Corresponding assembly: **`opcode rd rs rt`**
    - `GPR[rd] = GPR[rs] op GPR[rt]`
    - `PC = PC + 4 (use the next instruction)`

I-type Instructions – ALU ver.

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- `opcode`: there are immediate ALU instructions
  - `addi, addiu, andi, ori, xori, slti, sltiu, lui, …`
- Assembly and semantics
  - Corresponding assembly: **`opcode rt rs immediate`**
    - `GPR[rt] = GPR[rs] op sign-extend(immediate)`
      - `sign-extend(1000…000) = concat(11…1,1000…000)`
    - `PC = PC + 4 (use the next instruction)`
- What if you need to perform 32-bit immediate?
  - `lui at 0xABCD // store the upper 16 bits`
  - `ori at 0x1234 // store the lower 16 bits`

I-type Instructions – Memory ver.

opcode	rs	rt	offset
6-bit	5-bit	5-bit	16-bit

I-type

- opcode: there are immediate memory instructions
  - lw, lh, lhu, lb, lbu, sw, sh, sb, …
  - // lw indicates load, sw indicates store

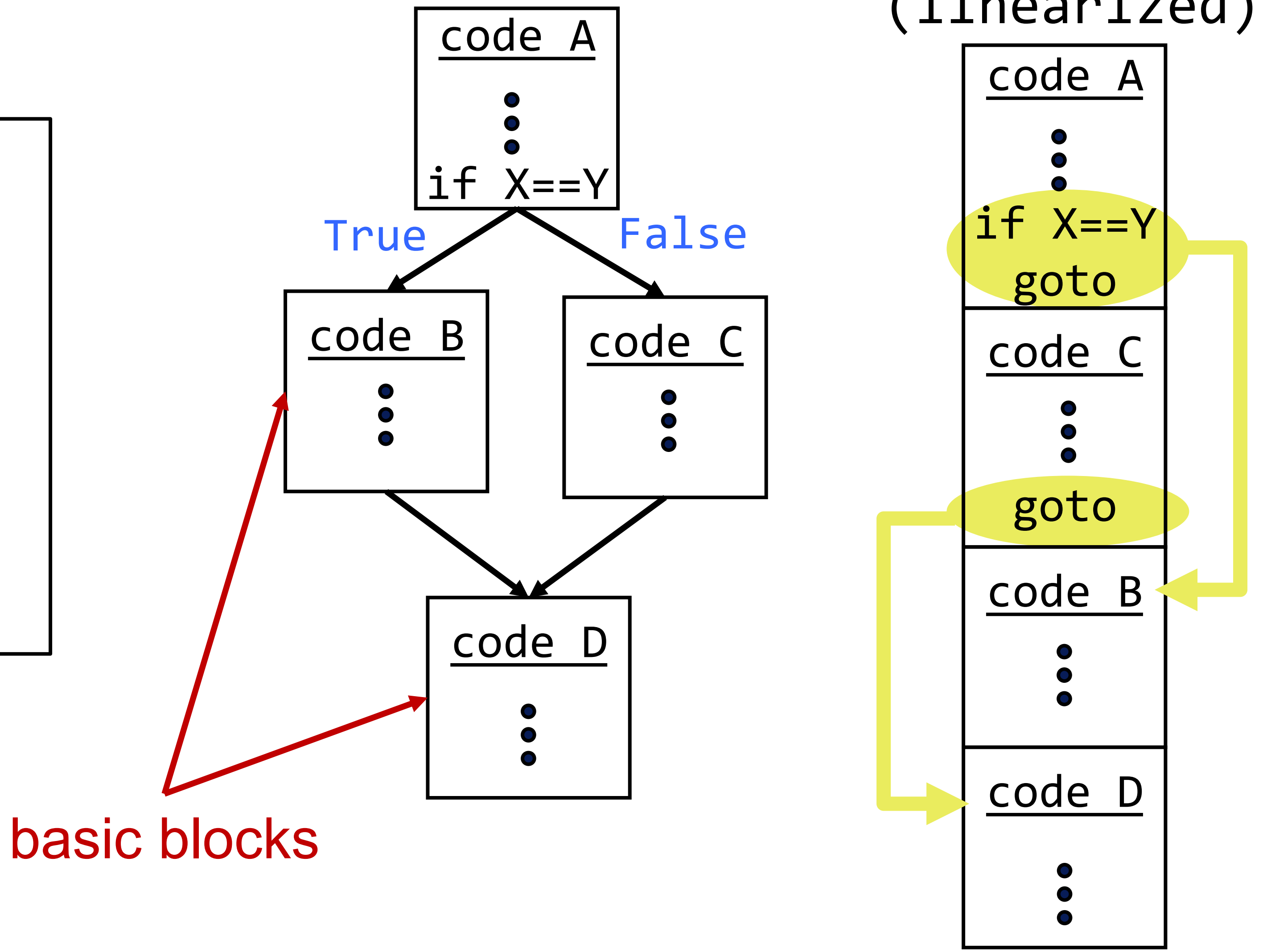
- Assembly and semantics
  - Corresponding assembler: **load/store rt offset(rs)**
    - GPR[rt] = MEM[GPR[rs] + sign-extend(offset)] // load
    - MEM[GPR[rs] + sign-extend(offset)]= GPR[rt] // store
    - PC = PC + 4 (use the next instruction)

Control Flow Instructions

Example code:

```
code A
if (X == Y){
        code B
}
else {
        code C
}
code D
```

Control Flow Graph — Assembly Code (linearized)



I-type Instructions – Control ver.

opcode	rs	rt	label
6-bit	5-bit	5-bit	16-bit

I-type

- `opcode`: there are eight immediate memory instructions
  - `bne, beq, …`
  - `branch equal or not equal …`
- Assembler and semantics
  - `Corresponding assembly:` **`beq rs rt label`**
    - `target = (PC + 4) + sign-extend(label) x 4` // word-aligned
    - `if (GPR[rs] == GPR[rt]) PC = target else PC = PC + 4`
  - If you want to jump more than 18 bits (i.e., 16 bit + 2 bit) ➔ Utilize J-type (in the following slides)

J-type Instructions

opcode	label	J-type
6-bit	26-bit	

- opcode: there are eight immediate memory instructions
  - j, jal

- Assembler and semantics
  - target = (PC+4)[31:28]x2^{28} $|_{\text{bitwise-or}}$ zero-extend(label) x 4
  - // use the first four bits of PC and append label x 4
  - Corresponding assembly: **j label**
    - PC = target
  - Corresponding assembly: **jal label**
    - GPR[ra] = PC + 4
      - save the next PC to ra (a dedicated register)
    - PC = target

R-type Instructions – Control ver.

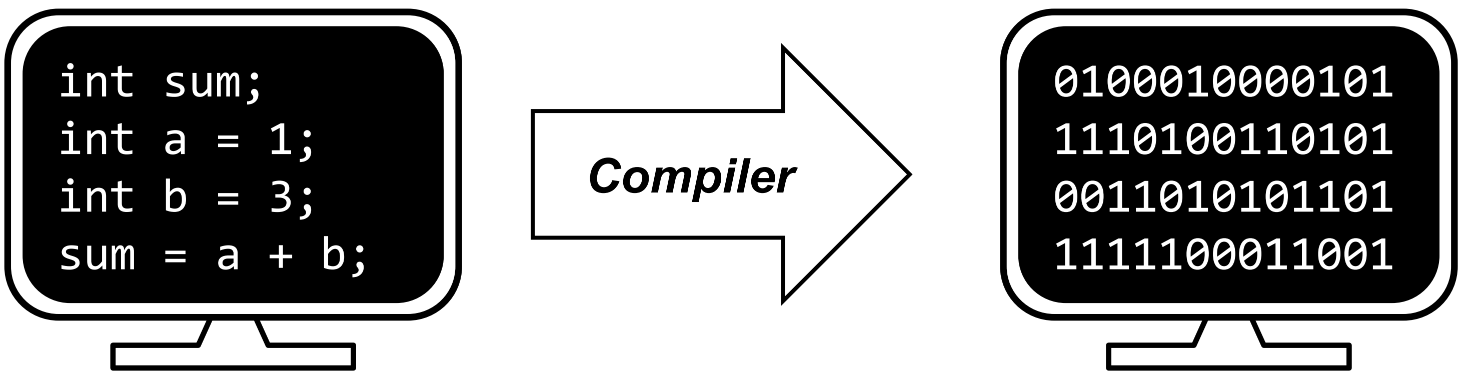
0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

- There exists an R-type control instruction
  - `jr`
- Generally used along with `jal label` (upon a function call)
- Assembler and semantics
  - Corresponding assembly: **`jr rs`**
    - `PC = GPR[rs]`

Filling the Gap

- **The overall compilation toolchain translates your program into a computer-executable form**
  - Your program is human-understandable language
  - But, computers can execute binary instructions
  - The compiler is there to fill the gap!

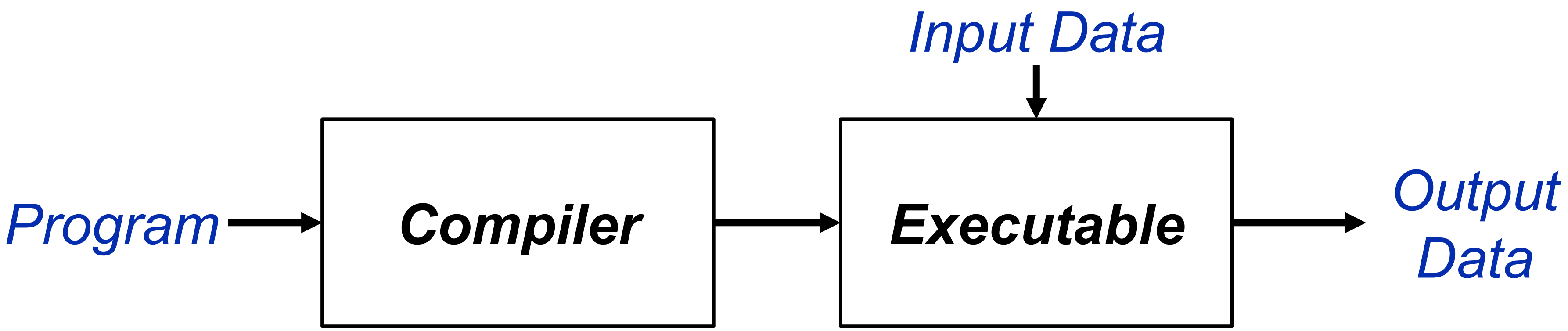


How you describe the program (Program)

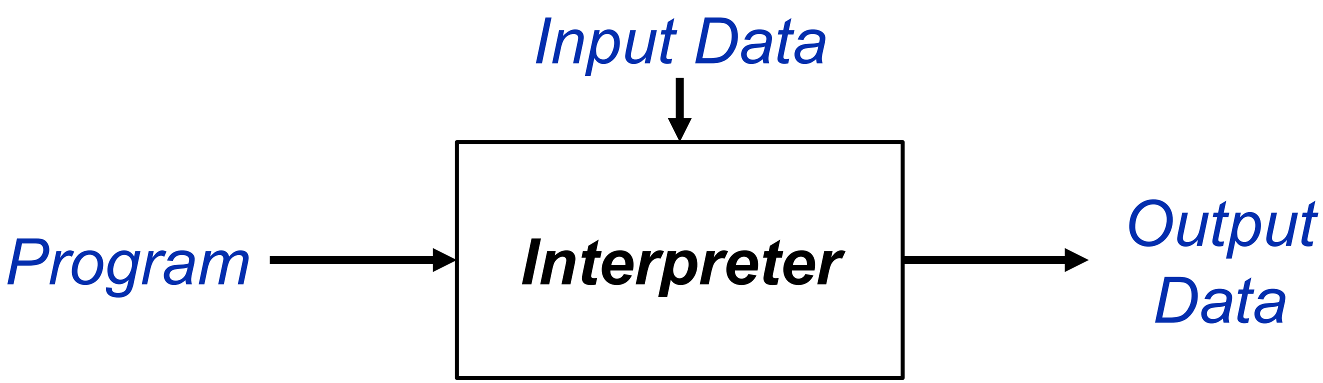
What computer understands (Executable)

Compiler vs. Interpreter

- **Compilers (The main focus)**



- **Interpreter**



Interpreter in Modern Processors

- Transmeta
  - Operates using a VLIW ISA
  - Utilizes **code morphing software (CMS)** to dynamically translate x86 instructions to Transmeta VLIW instructions

- Java
  - JVM translates the java bytecode into the ISA

- x86 Processors
  - x86 CPU uses a uop (not CISC) instructions internally
  - The hardware translates the x86 CISC instructions to RISC-style uops

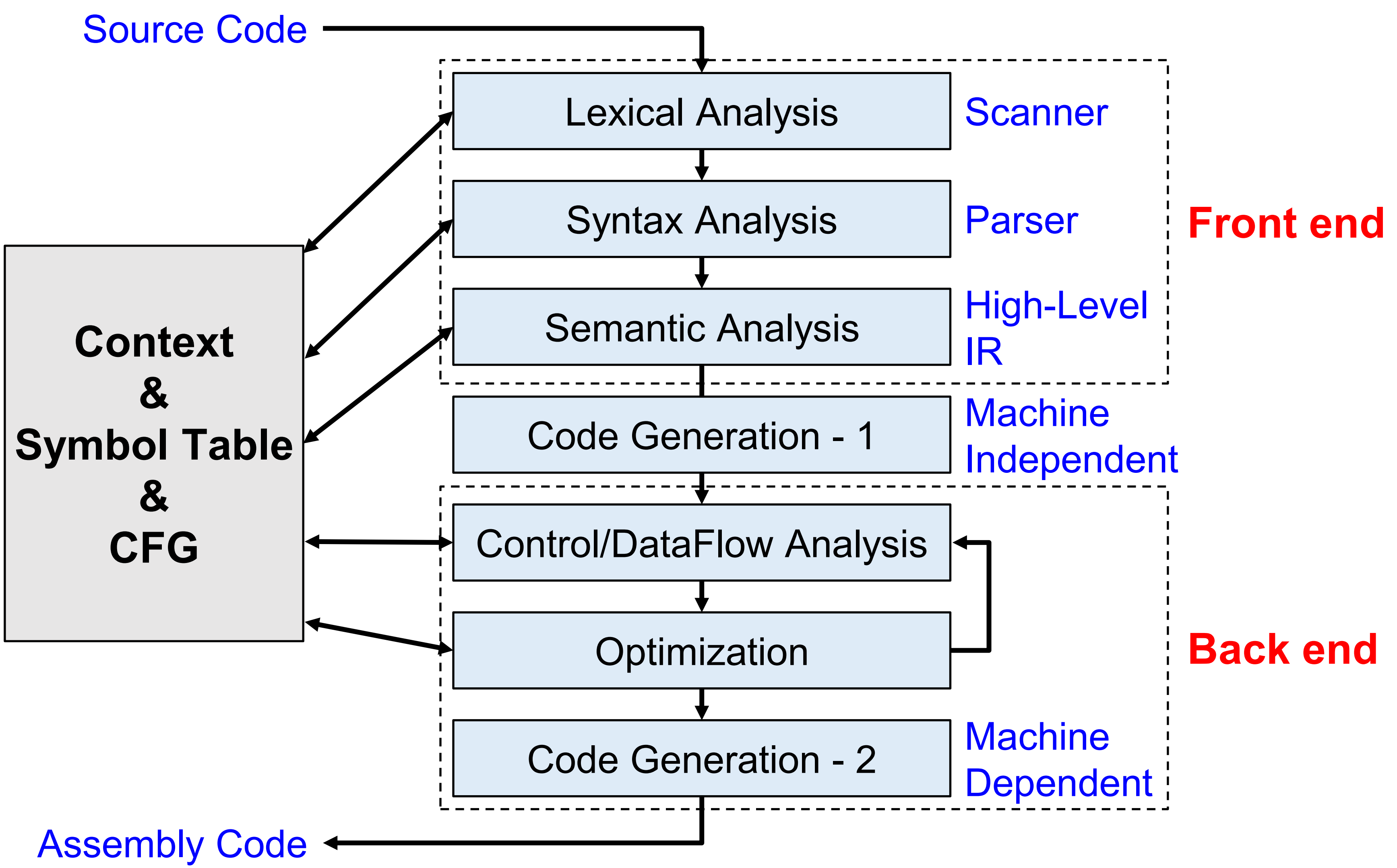
Super fun topics, but not the coverage of this class

Compiler

- **Front-End: Lets you program with high-level languages**
  - In 1970s, you had to use punch cards for programming
  - Now, we are in the surge of diverse programming languages
    - Fortran, C, C++, Java, Python, R, Tex, Html, …

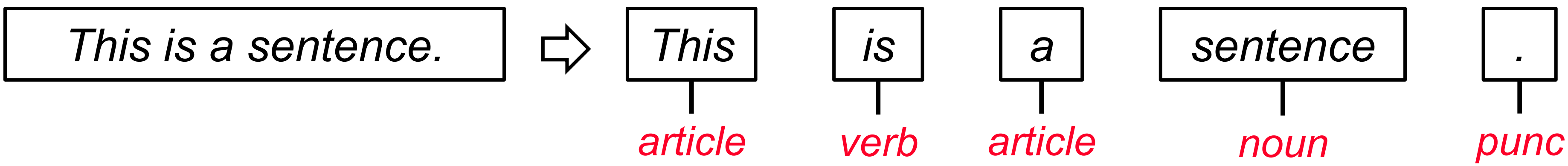
- **Back-End: Let's you program without considering hardware-specific or straightforward optimization issues**
  - The internal computer architectures are hard for programmers to fully understand
  - The compiler optimizes the redundant codes

General Structure of a Modern Compiler

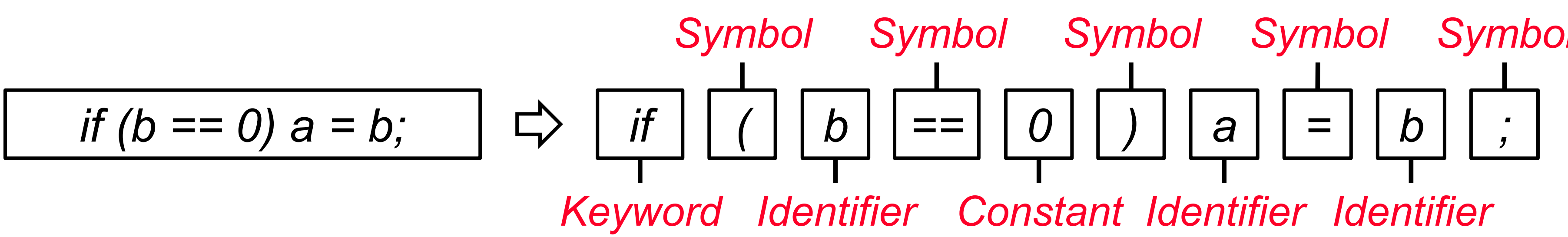


Lexical Analysis (Scanner)

- **Language: Recognizing words from sentences**



- **Program: Dividing programs into “tokens”**

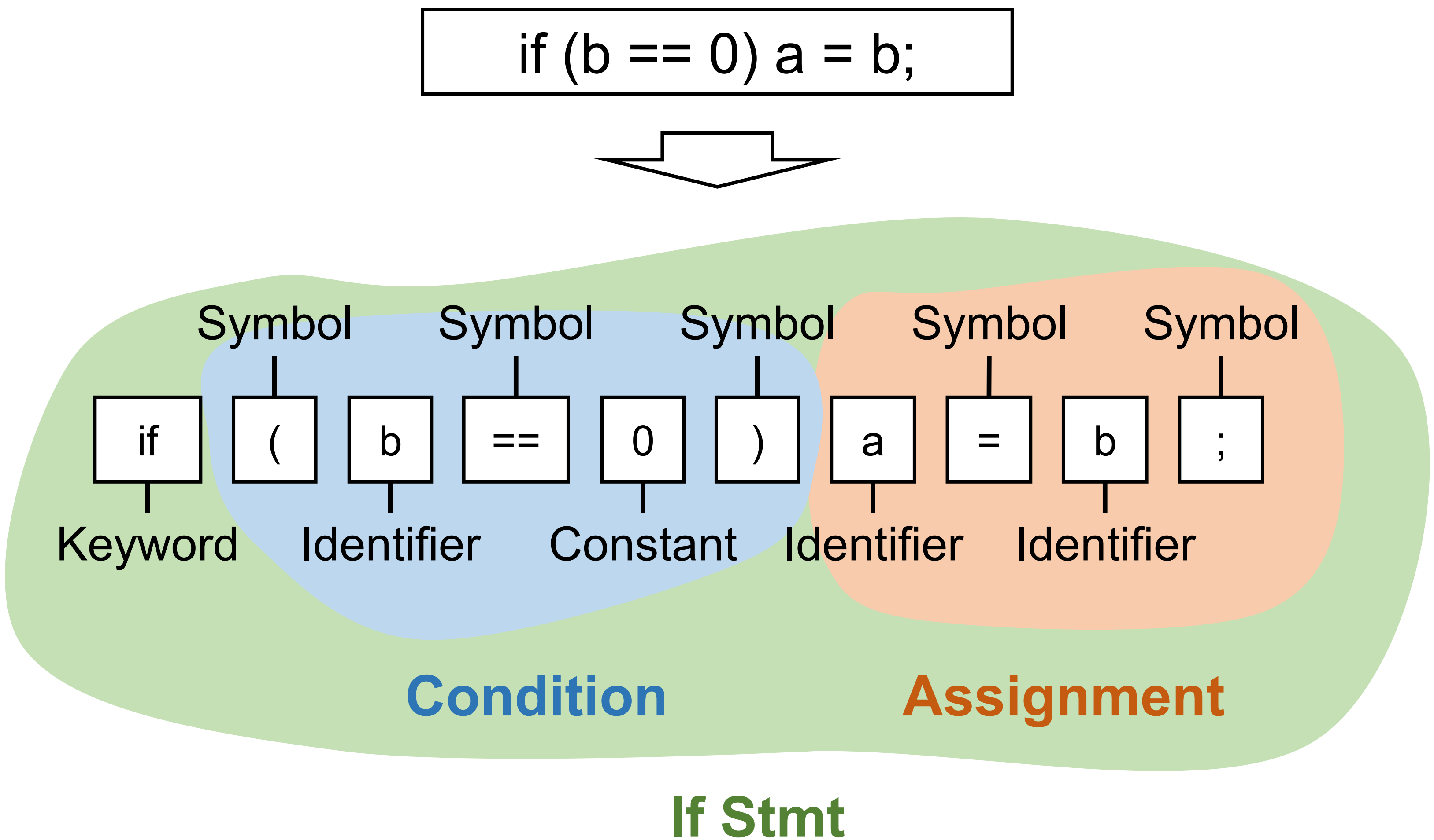


Lexical Analysis (Scanner)

- Extracts and identifies lowest-level lexical elements
  - Reserved words: for, if, switch
  - Identifiers: “i”, “j”, “table”
  - Constants: 3.14159, 17, “%d\n”
  - Punctuation symbols: “(“,”)”, “,”, “+”
- Removes non-grammatical elements – i.e. spaces, comments
- Implemented with a Finite State Automata (FSA)
  - Set of states – partial inputs
  - Transition functions to move between states

Syntax Analysis (Parser)

- **Program: Determines the relationship between tokens**



Syntax Analysis (Parser)

- **Checks the program for syntactic correctness**
  - Framework for subsequent semantic processing
- **Lots of variations**
  - Hand-written recursive descent
  - Table-based (Top-down vs. Bottom-up)

Semantic Analysis

- **Should support several distinct actions**
  - Check identifier definitions and validate the correctness
  - Type checking
  - Declaration scopes
  - Disambiguate overloaded operators
  - Translate from source to intermediate representation (IR)
  - Etc

Optimization

- **Optimize code quality**
- **There are several optimization opportunities**
  - Identify constant values and compute them in advance
  - Remove unused variables
  - Compute loop-invariant variables
  - Remove recomputed variables
  - Etc

Code Generation

- **Map machine independent intermediate representation to the target architecture**

- **Virtual to physical binding**
  - Instruction selection: best machine opcodes to implement generic opcodes
  - Register allocation: infinite virtual registers to N physical registers
  - Assembly emission

- **Machine assembly is our output, assembler, linker take over to create binary**

Question?