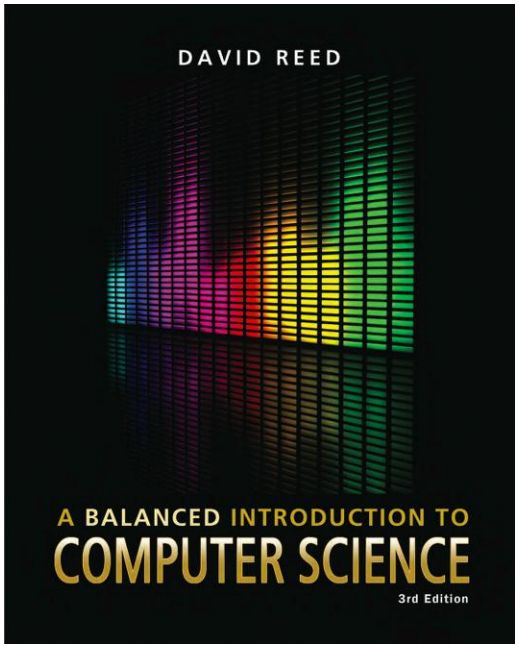
A Balanced Introduction to Computer Science, 3/E

David Reed, Creighton University

ISBN 978-0-13-216675-1

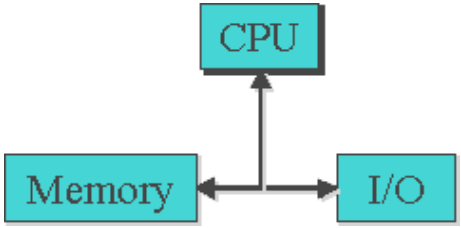


Chapter 14
Inside the Computer - the von Neumann Architecture

CPU Subunits and Datapath

recall the von Neumann architecture

- Central Processing Unit (CPU)
- Input/Output devices
- Memory



the CPU acts as the brain of the computer

- it obtains data and instructions from memory
- it carries out instructions
- it stores results back to memory

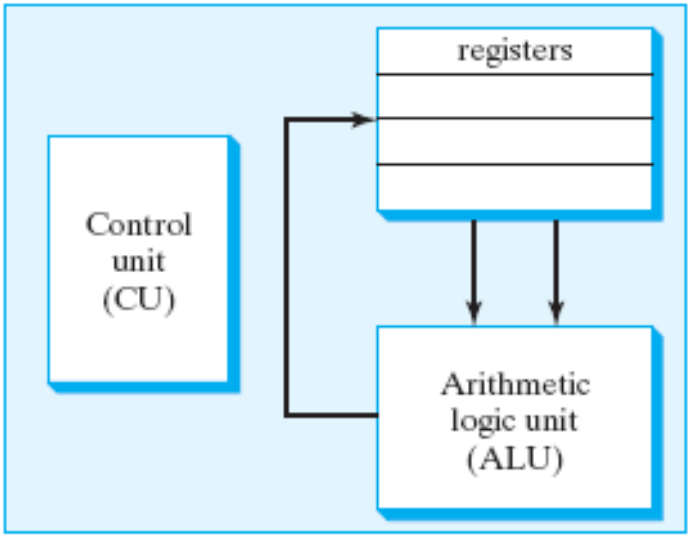
the program instructions that are executed by the CPU are in the *machine language* of that CPU

- users can write programs in a high-level language, but those high-level instructions must be translated into machine language before executing
  - even programs that exhibit complex behavior are specified to the CPU as sequences of simple machine-language commands
  - the CPU can execute these instructions at such a high speed that complex programmatic behavior is achieved

CPU Subunits

the CPU is comprised of several subunits, each playing a specific role in processor functioning

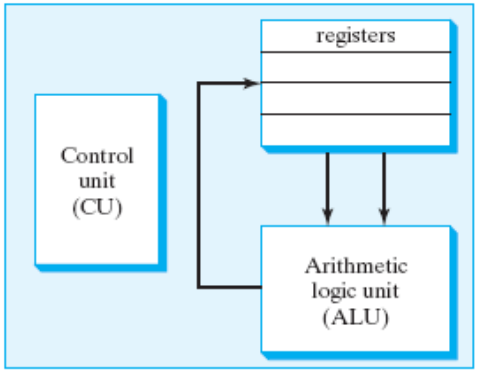
- *Arithmetic Logic Unit (ALU)*: circuitry that performs the actual operations on data (addition, subtraction, bit manipulations)
- *Registers*: memory locations built into the CPU (to provide fast access)
  - the transfer of data between registers and main memory occurs across a collection of wires called a *bus*
- *Control Unit (CU)*: circuitry in charge of fetching data and instructions from main memory, as well as controlling the flow of data between registers and the ALU



CPU Datapath Cycles

the path that data follows within the CPU, traveling along buses from registers to the ALU and then back to registers, is known as the *CPU datapath*

- a single rotation around the CPU datapath is referred to as a *CPU datapath cycle*



recall earlier that we defined CPU speed as measuring the number of instructions that a CPU can carry out per second

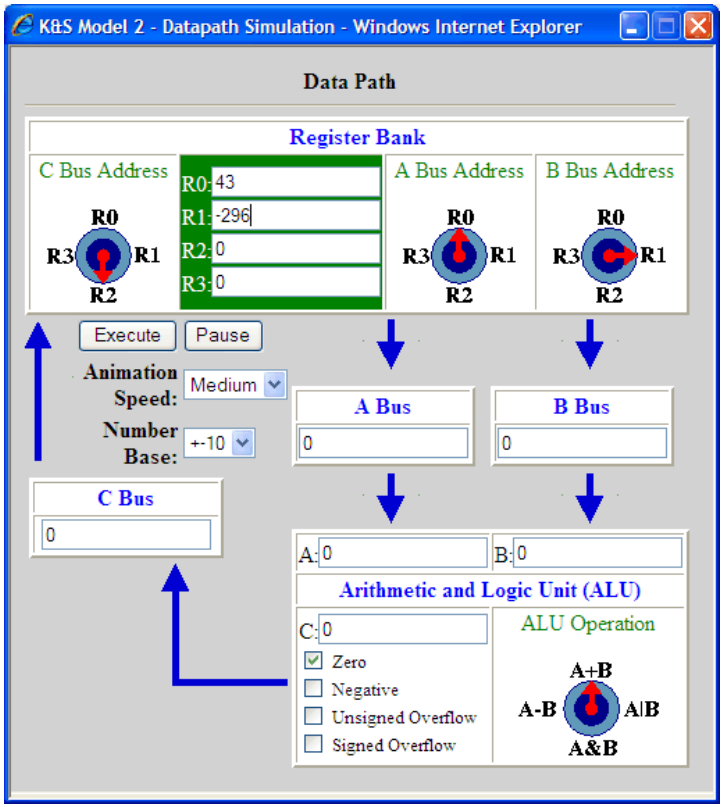
- CPU speed = the number of CPU cycles per second
  - e.g., a 2.5GHz CPU can perform 2.5 billion cycles per second

- CPUs cannot be compared solely on the basis of their speeds
  - since CPUs have different instruction sets, one CPU may be able to complete more complex tasks in a single cycle
  - so, a slower CPU with a richer instruction set may complete some tasks faster

Datapath Simulator

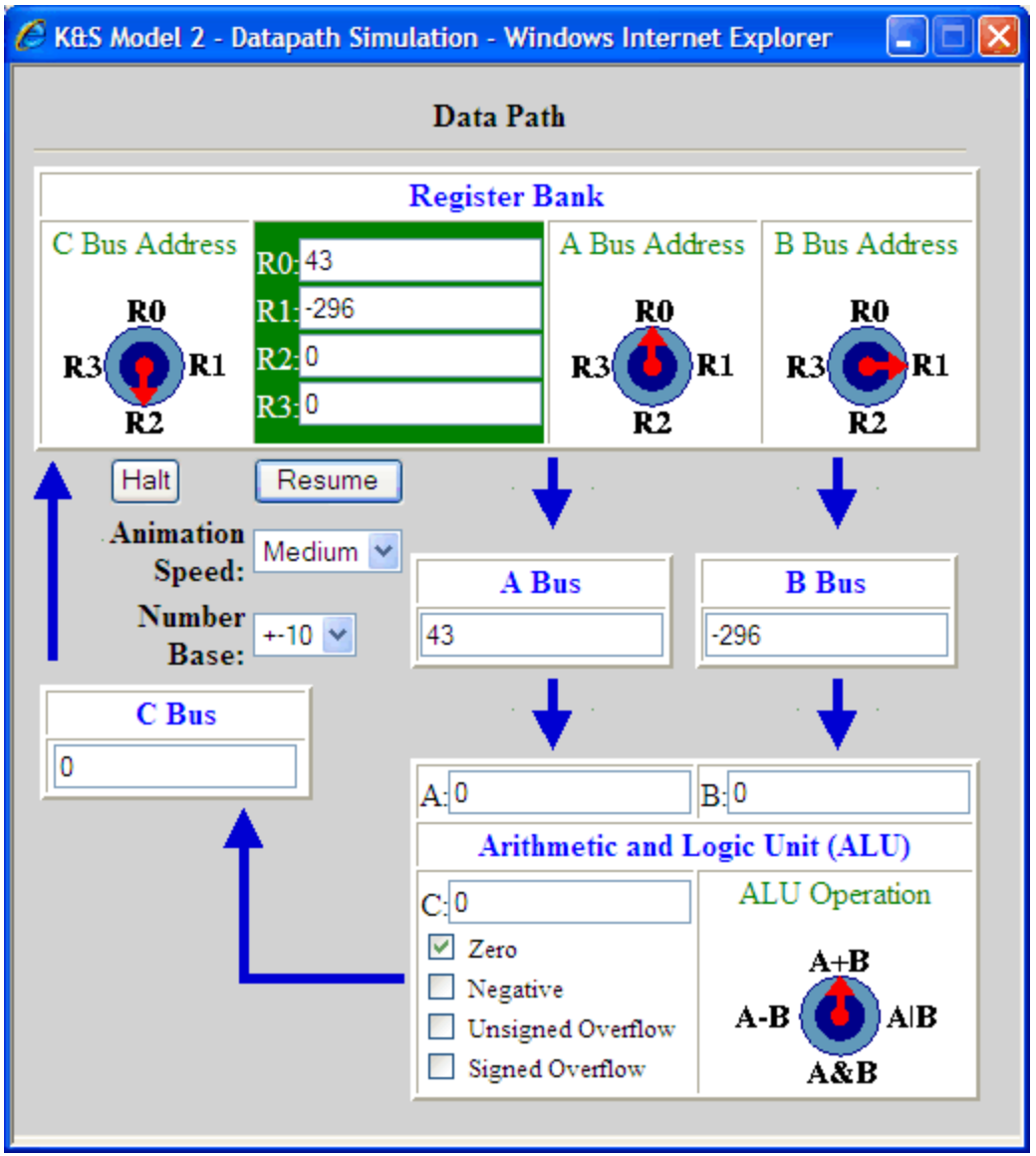
accompanying the text is a datapath simulator

- a.k.a. the *Knob & Switch Computer*
- developed by Grant Braught at Dickinson College
- allows the user to enter data directly into the register boxes and configure the CPU hardware by clicking on knobs and switches



- ✓ by clicking on the knobs labeled *A Bus Address* and *B Bus Address*, the user can select which registers will be accessed
  - here, registers R0 and R1 are selected
- ✓ by clicking on the knob labeled *ALU Operation*, the user can select which operation will be performed on the specified register data
  - here, the addition operation (A+B) is selected
- ✓ by clicking on the knob labeled *C Bus Address*, the user can select which register the result will be written to
  - here, register R2 is selected

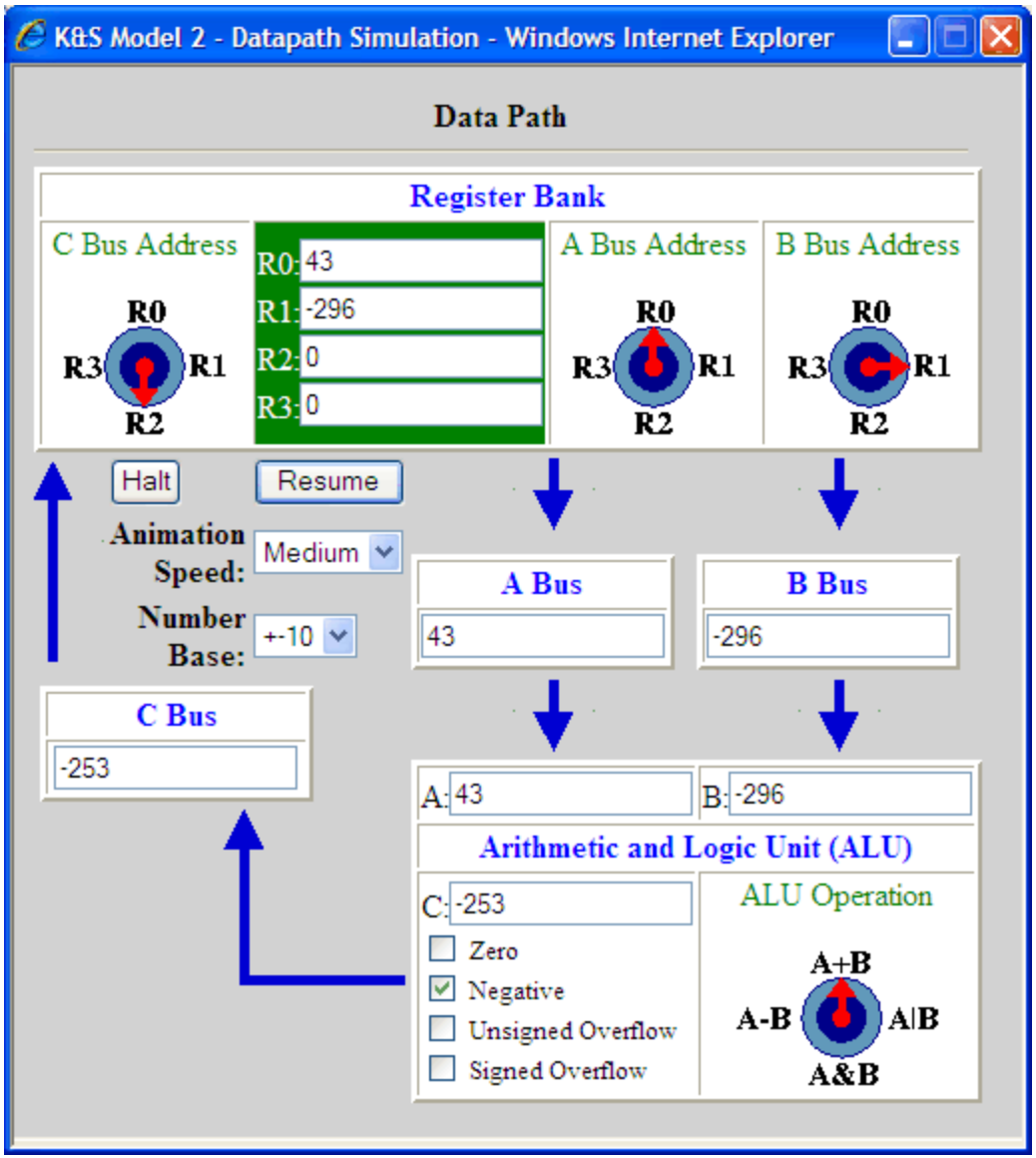
Datapath Simulator (cont.)



when the user clicks on the Execute button, a CPU datapath cycle is executed with the given settings

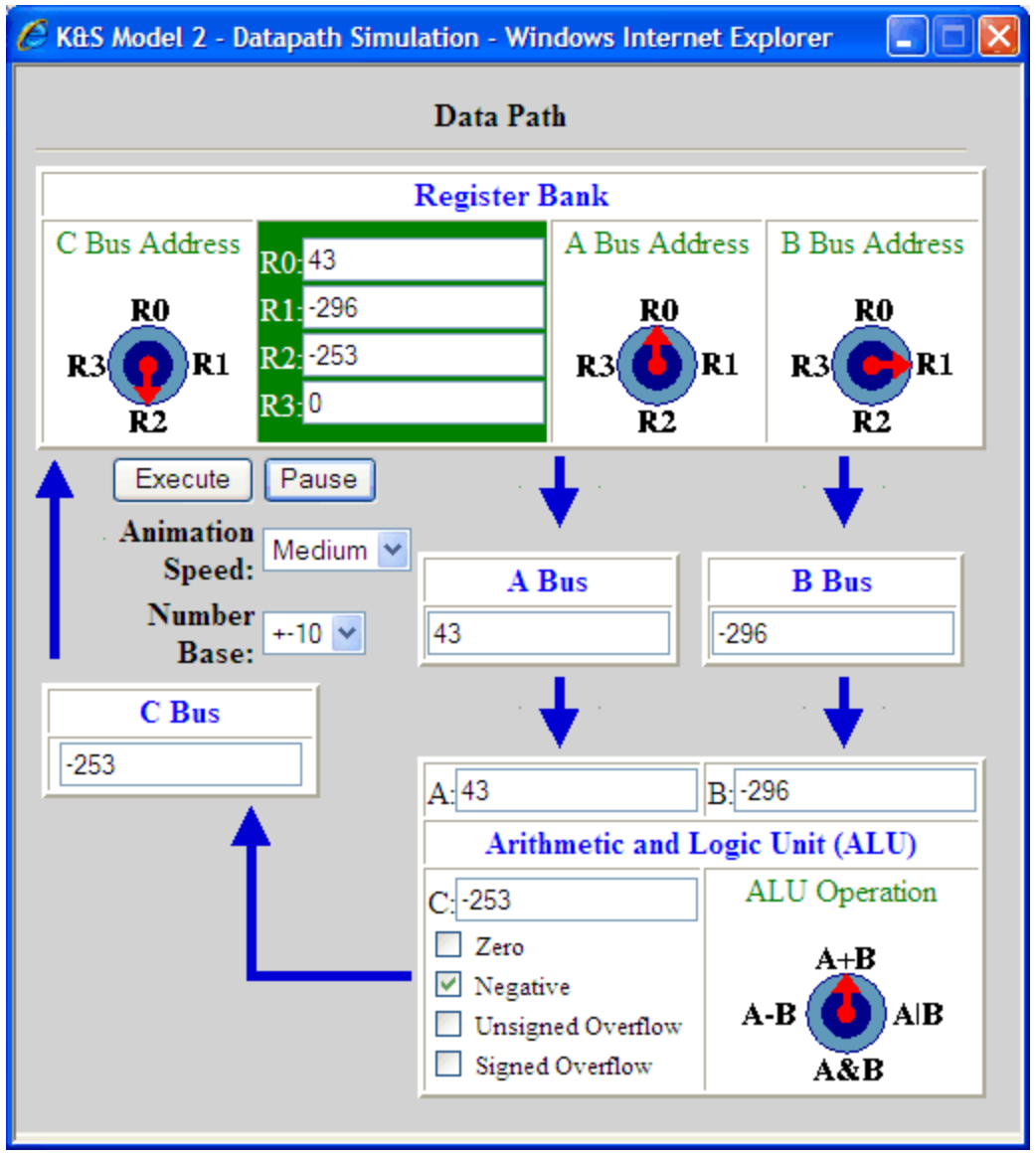
- data can be observed as it moves around the datapath
- in the 1st step, data values are retrieved from the specified registers (R0 and R1) and passed along the A and B buses, respectively

Datapath Simulator (cont.)



- data can be observed as it moves around the datapath
- in the 2nd step, the data values from the A and B buses are operated on by the ALU, with the result passed along the C bus

Datapath Simulator (cont.)



- data can be observed as it moves around the datapath
- in the 3rd step, the data on the C bus is written to the specified register (R2)

the end result of the datapath cycle is that the contents of two registers are added and the result is stored in a different register

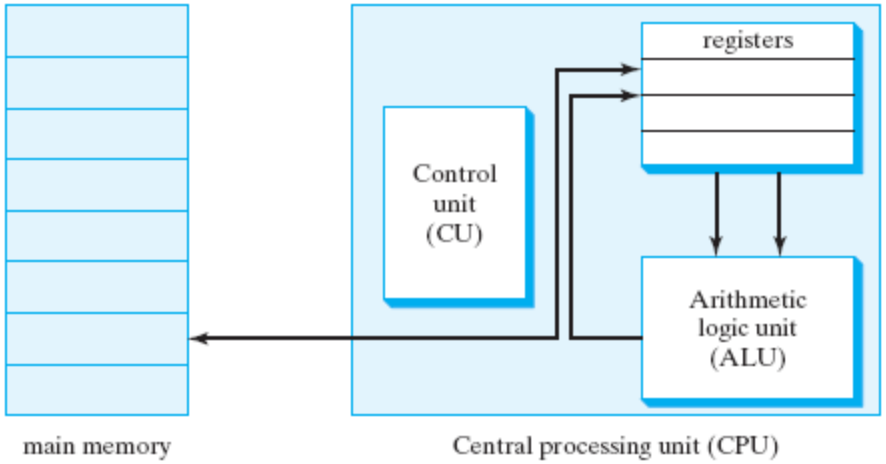
CPU and Main Memory

the CPU datapath describes the way in which a computer manipulates data stored in registers

- but, it does not explain how data gets into registers in the first place
- or, how the results of ALU operations are accessed outside the CPU

both of these tasks involve connections between the CPU and main memory

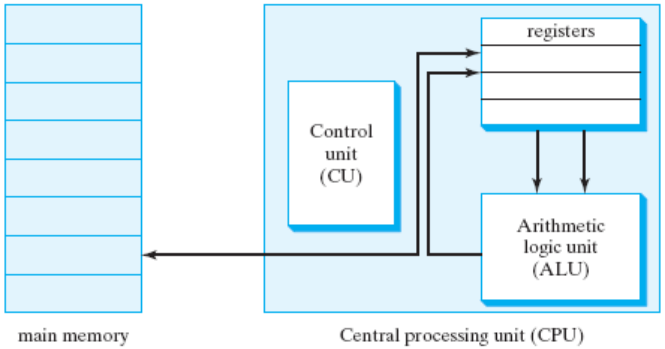
- all active programs and data are stored in the main memory of a computer
- main memory is comprised of a large number of memory locations, with each location accessible via an address
- a bus connects main memory to the CPU, enabling the computer to copy data and instructions to registers, and then copy the results of computations back to main memory



Main Memory Data Transfer

as a program is executed, the control unit processes the program instructions and identifies which data values are needed to carry out the specified tasks

- the required values are then fetched from main memory along the main memory bus, loaded into registers, and used in ALU operations



note: data must be copied from main memory into registers BEFORE it can be operated on

- in practice, the data transfer takes much longer than a single datapath cycle (due mainly to the distance that must be covered)
- modern CPUs compensate by fetching multiple instructions at once – ideally, the CPU will predict which instructions will be needed next and prefetch them

Data Transfer Example

example: suppose you had a file containing 1,000 numbers and wanted to compute the sum of those numbers

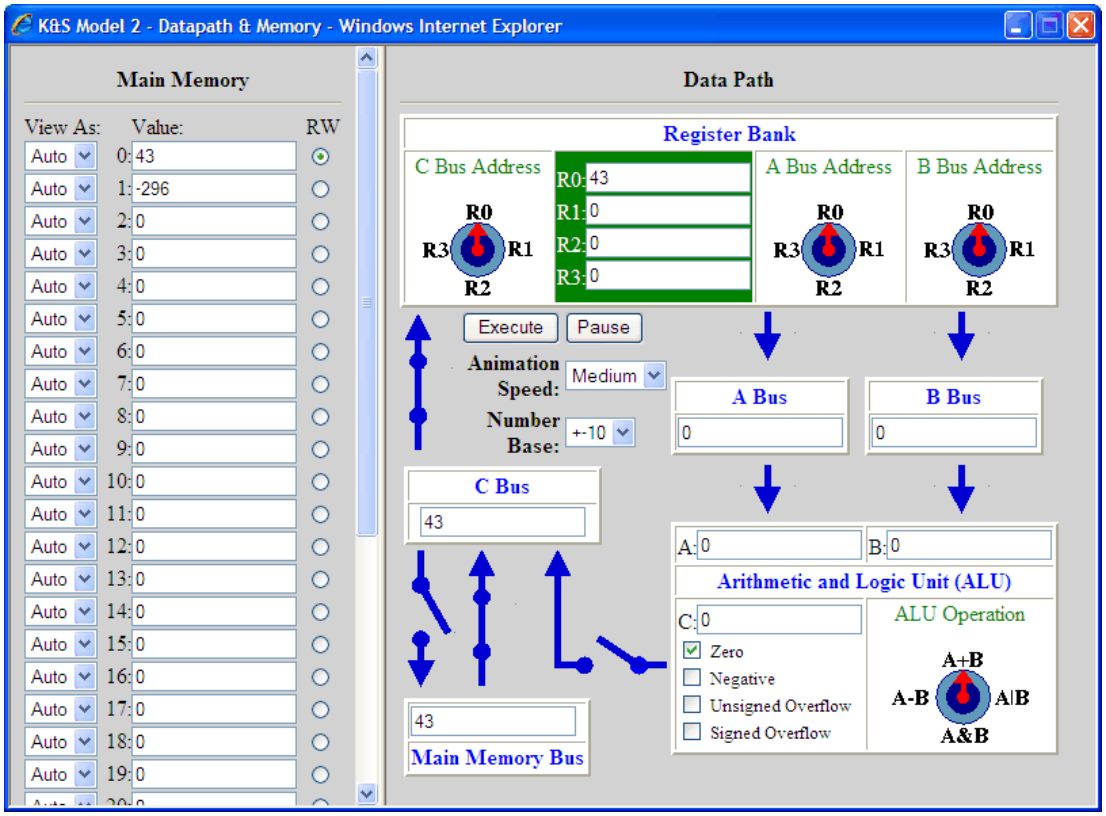
- first, need to load the data from the file into main memory — for example, at memory locations 500 through 1499
- then, the control unit would carry out the following steps to add the numbers and store the resulting sum back in main memory

1. Initialize one of the registers, say R0, to 0. This register will store the running total of the numbers.
2. For each number stored at memory addresses 500 through 1499:
   a. Copy the number from main memory to another register, say R1.
   b. During one cycle around the CPU datapath, add the contents of R0 and R1 and store the result back in R0.
3. When all the numbers in the file have been processed, the value in R0 will represent their sum. This value can then be copied back to a main memory location.

Datapath w/ Memory Simulator

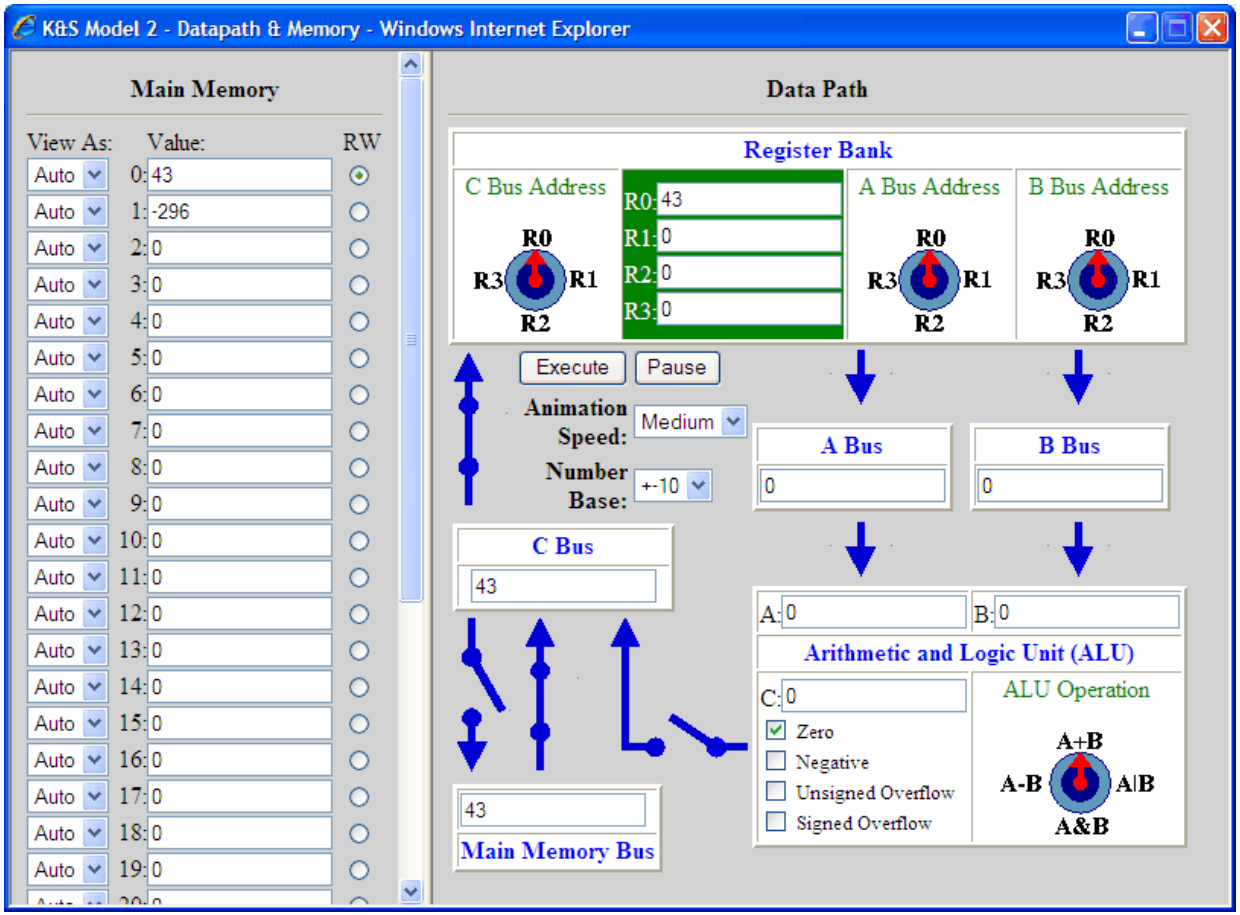
we can extend the datapath simulator to integrate main memory

- main memory is displayed as a series of boxes, labeled with addresses 0-31
- a Main Memory Bus is added to transfer data to and from main memory
- buses have switches that open and close when the user clicks on them



- ✓ by selecting the radio button labeled R/W in the Main Memory partition, the user can select a memory address
  - here, address 0 is selected
- ✓ by clicking on the Main Memory Bus and C Bus, they can be opened/closed
  - here, the buses are configured to allow data to flow from main memory to the register
- ✓ the other components of the datapath are configured as before

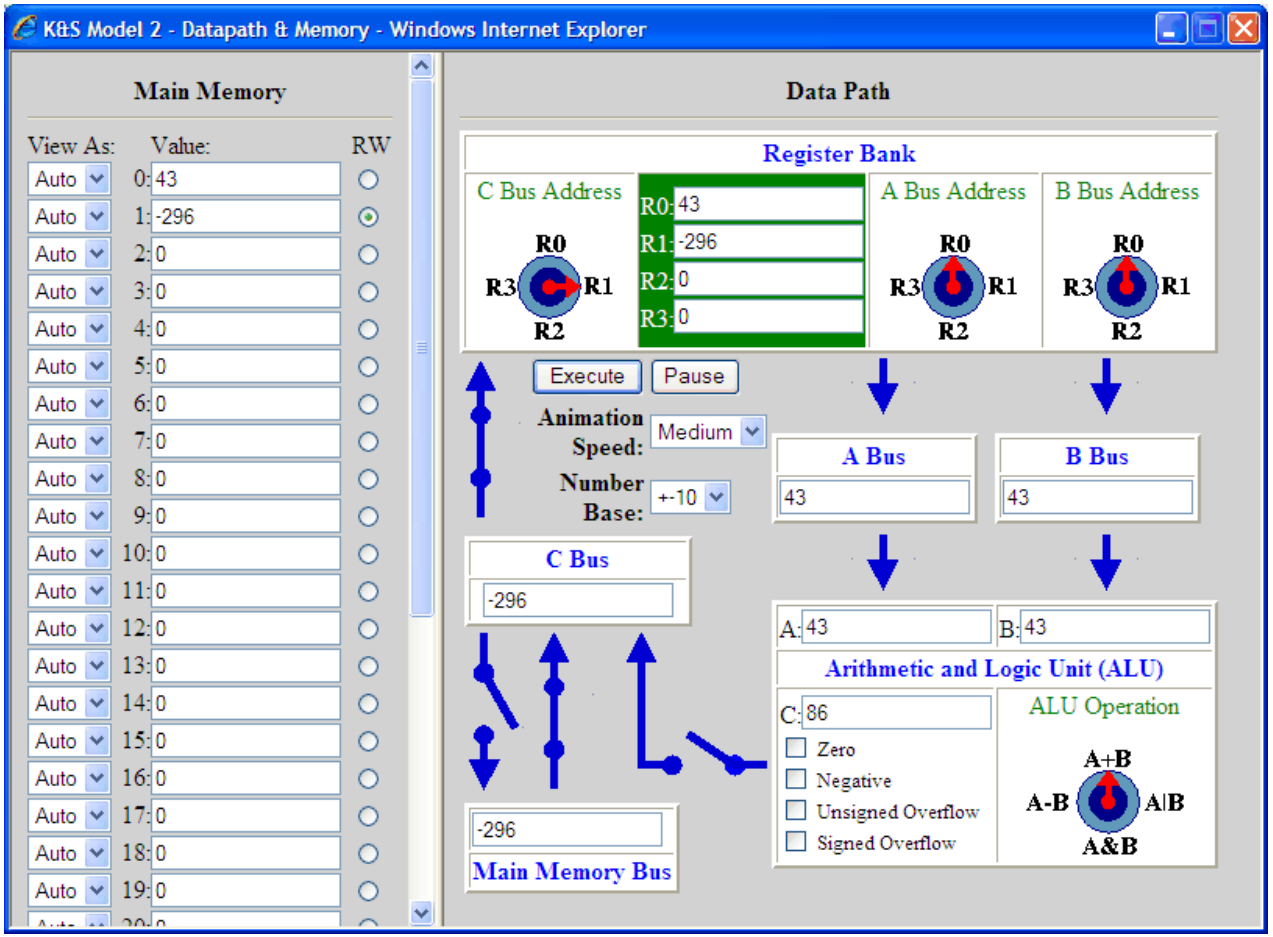
Memory Simulator (cont.)



for example, suppose we wanted to add the contents of address 0 and address 1, and store the result in address 2

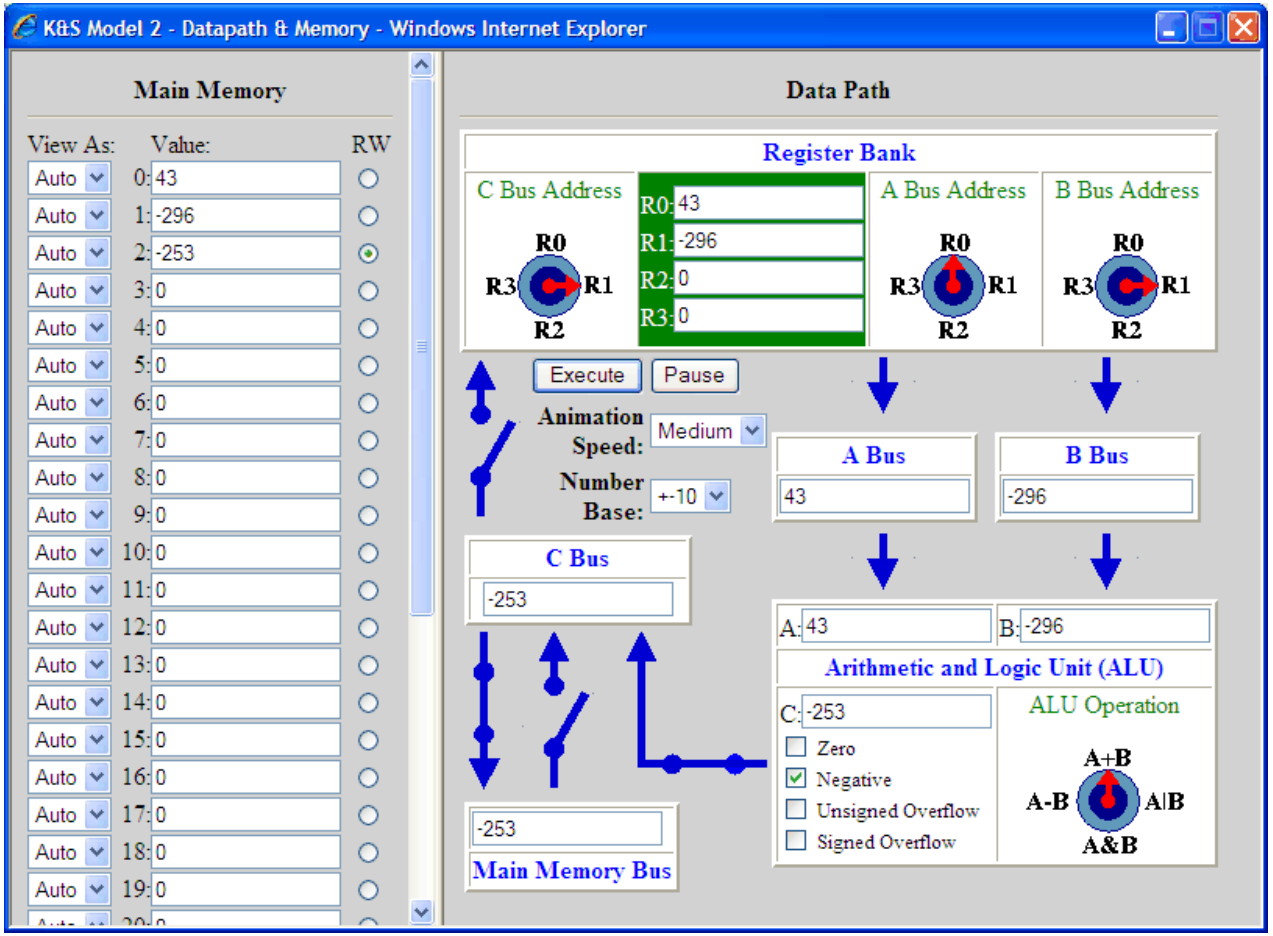
- in the 1st step, copy the data from address 0 into a register, say R0
- note: the A Bus, B Bus, and ALU settings are irrelevant since the C Bus is disconnected

Memory Simulator (cont.)



- in the 2nd step, similarly copy the data from address 1 into a register, say R1
- note: when the simulator executes, there is a noticeable (and intentional) delay as data moves on the Main Memory Bus

Memory Simulator (cont.)



- in the 3rd step, the contents of R0 and R1 are added and the result is sent directly to address 2
- in some CPUs, this would require 2 steps: send the result to a register, then copy the register to main memory

Stored-Program Computer

the last component of the CPU is a fully functioning, automatic control unit

- recall that in the simulations, tasks were defined by selecting registers and ALU operations via knobs, switches, and buses
- the idea behind a stored-program computer is that tasks can be represented as instructions stored in main memory along with data
- the control unit must fetch, decode, and execute those instructions

recall: each type of CPU has its own machine language for representing instructions

- for example, we could define an instruction for our simulator by specifying the knob and switch settings for the datapath

```
A Bus = R0          ALU Switch = closed
B Bus = R1          MMIn Switch = open
ALU = A+B           MMOut Switch = open
C Bus = R2          C Switch = closed
```

- since real world CPUs contain an extremely large number of physical components, a more compact representation is required
- machine language instructions are represented as binary strings, where the bits correspond to hardware settings

Simulator Machine Language

Operation	Machine-Language Instruction	Example
Add contents of two registers, store result in another register e.g., `R0 = R1 + R2`	`1010000100 RR RR RR`	`1010000100 00 01 10` will add contents of R1 (01) and R2 (10), then store the result in R0 (00)
Subtract contents of two registers, store result in another register e.g., `R0 = R1 - R2`	`1010001000 RR RR RR`	`1010001000 00 01 10` will take contents of R1 (01), subtract R2 (10), then store the result in R0 (00)
Load contents of memory location into register e.g., `R3 = M[5]` (11)	`100000010 RR MMMMM`	`100000010 11 00101` will load contents of memory location 5 (00101) into R3
Store contents of register in memory location e.g., `M[5] = R3`	`100000100 RR MMMMM`	`100000100 11 00101` will store contents of memory location 7 (00101) in R3 (11)
Move contents of one register into another register e.g., `R1 = R0`	`100100010000 RR RR`	`100100010000 01 00` will move contents of R0 (00) to R1 (01)
Halt the machine	`1111111111111111`	N/A

1st two instructions – select ALU operation and registers to operate on

next 3 instructions – control flow of data between main memory and datapath

last instruction – mark the end of instruction sequence

Control Unit

the control unit fetches each machine language instruction from memory, interprets its meaning, carries out the specified CPU cycle, and then moves on to the next instruction

- since instructions and data are both stored in memory, the control unit must recognize where a sequence of instructions begins and ends
- for our simulator, we will assume that programs start at address 0 and end with a HALT instruction
  - in real computers, the operating system must keep track of program locations

Fetch-Execute Algorithm carried out by the control unit:

1. Initialize PC = 0.
2. Fetch the instruction stored at memory location PC, and set PC = PC + 1.
3. As long as the current instruction is not the HALT instruction, repeatedly:
   a. Decode the instruction—that is, determine the CPU hardware settings required to carry it out.
   b. Configure the CPU hardware to match the settings indicated in the instruction.
   c. Execute a CPU datapath cycle using those settings.
   d. When the cycle is complete, fetch the next instruction from memory location PC, and set PC = PC + 1.

Program Execution

example: machine language program for adding two numbers in memory

```
0: 1000000100000101      // load memory location 5 into R0
1: 1000000100100110      // load memory location 6 into R1
2: 1010000100100001      // add R0 and R1, store result in R2
3: 1000001001000111      // copy R2 to memory location 7
4: 1111111111111111      // halt
5: 0000000000001001      // data to be added: 9
6: 0000000000000001      // data to be added: 1
7: 0000000000000000      // location where sum is to be stored
```

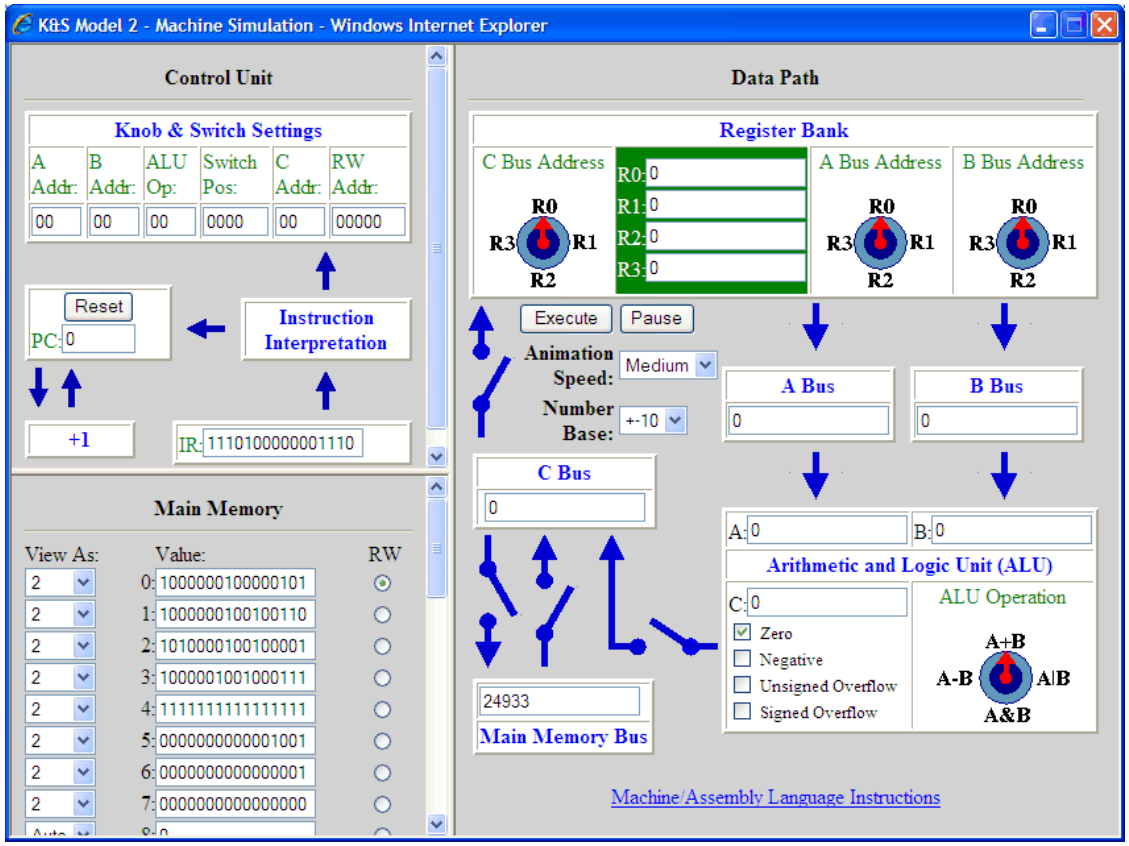
program execution:

1. program counter is initialized: PC = 0
2. instruction at memory location 0 is fetched, PC is incremented to 1
3. instruction is decoded and CPU cycle is executed: *load contents of memory location 5 into RO*
4. next instruction is fetched at location 1, PC is incremented to 2
5. instruction is decoded and CPU cycle is executed: *load contents of memory location 6 into R1*
6. next instruction at memory location 2 is fetched, PC is incremented to 3
7. instruction is decoded and CPU cycle is executed: *add contents of R0 and R1, store result in R2*
8. next instruction at location 3 is fetched, PC is incremented to 4
9. instruction is decoded and CPU cycle is executed: *copy contents of R2 into memory location 7*
10. next instruction at location 4 is fetched, PC is incremented to 5
11. instruction is decoded and CPU cycle is executed: *halt instruction → program ends*

Complete Computer Simulator

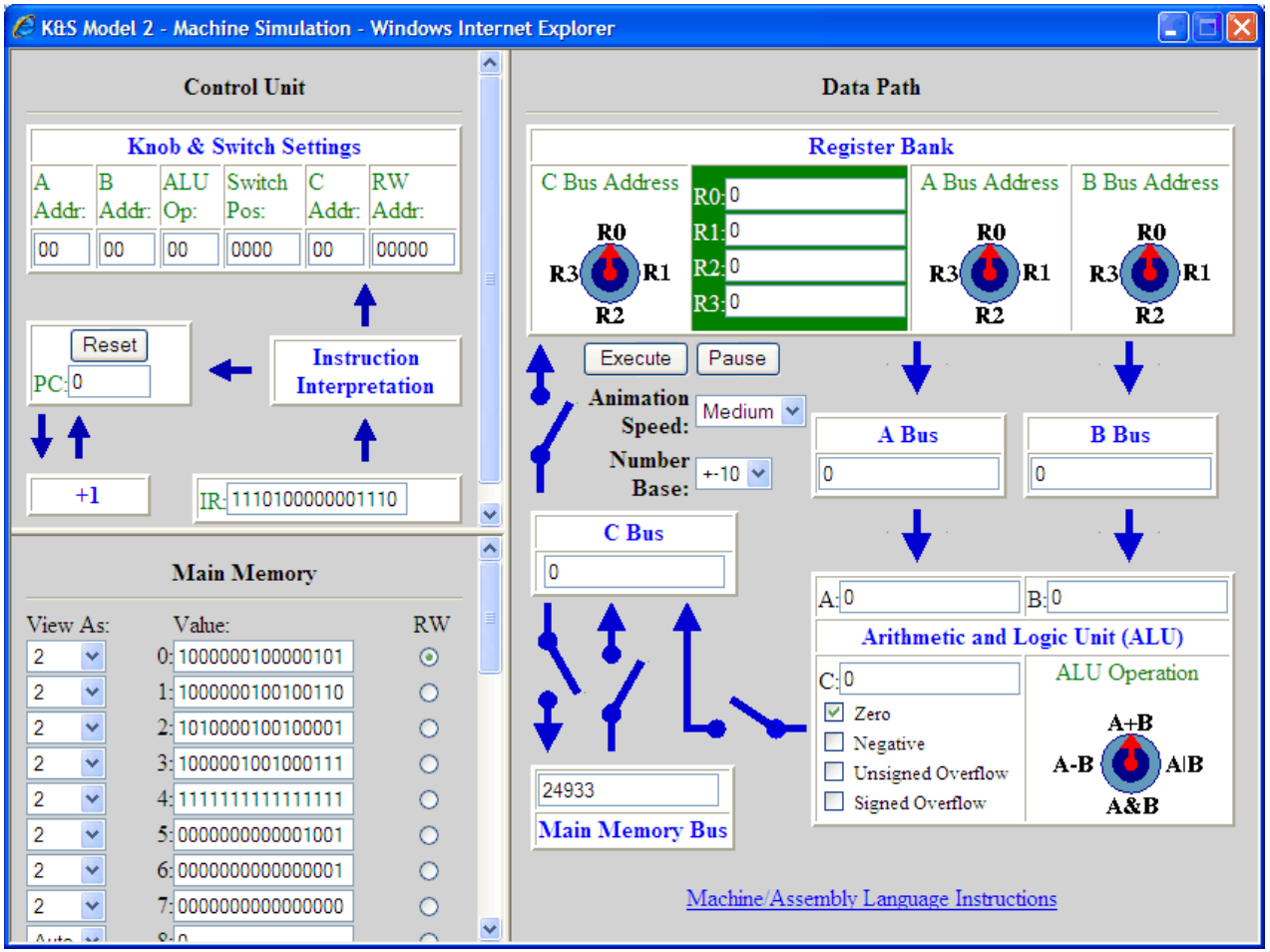
we can extend the simulator to integrate a control unit, containing

- Program Counter (PC) for keeping track of the next statement
- Instruction Register (IR): for storing fetched instructions



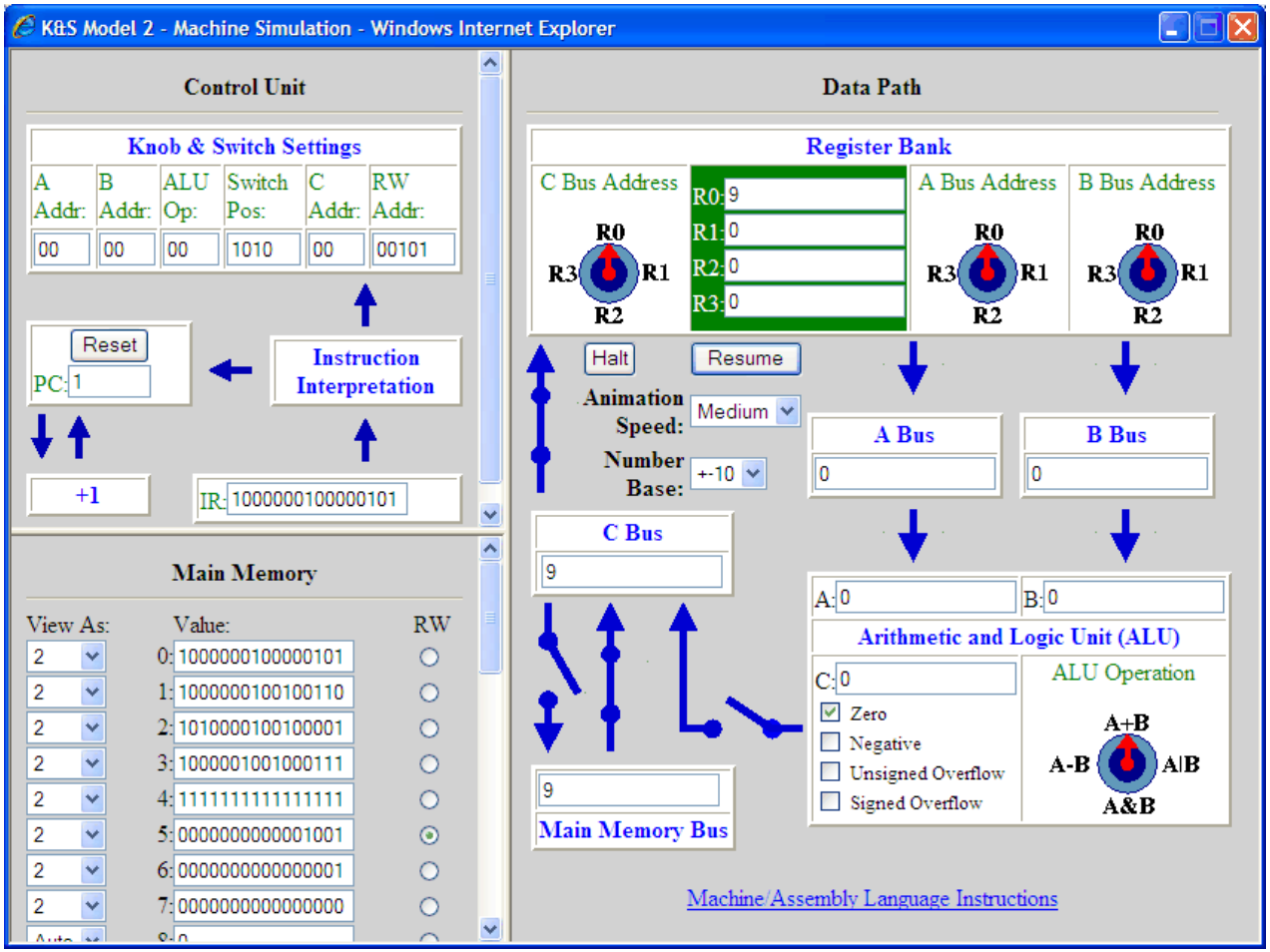
- ✓ the user can enter a sequence of instructions directly into memory, starting at address 0 and ending with a Halt instruction
- ✓ any data to be acted upon is stored in additional memory locations
- ✓ when the user clicks execute, the program instructions are executed in sequence

Computer Simulator Example



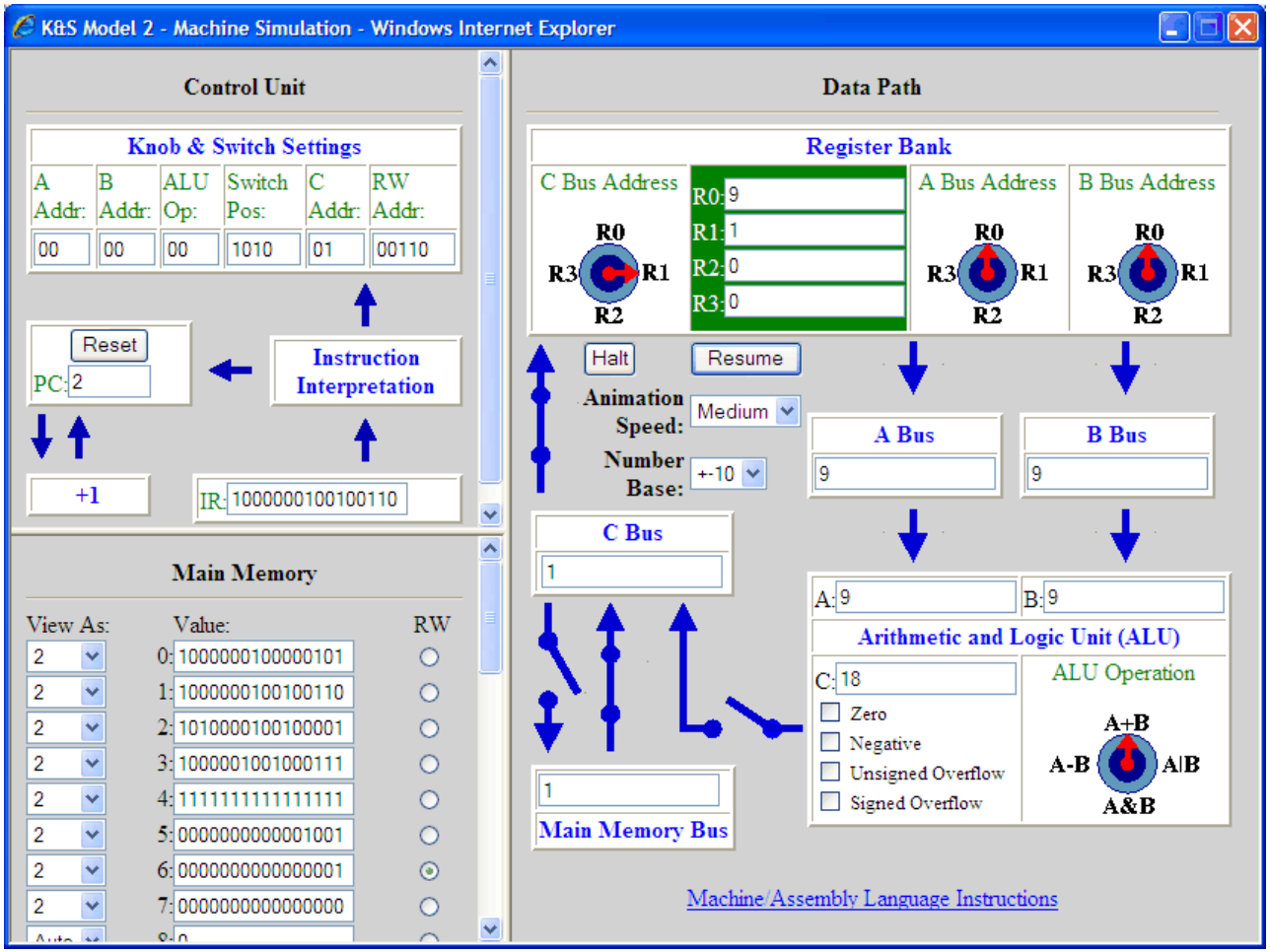
- here, the program consists of 5 instructions, stored in addresses 0 – 4
- data values are stored in addresses 5 & 6
- before execution, the PC = 0

Simulator Example (cont.)



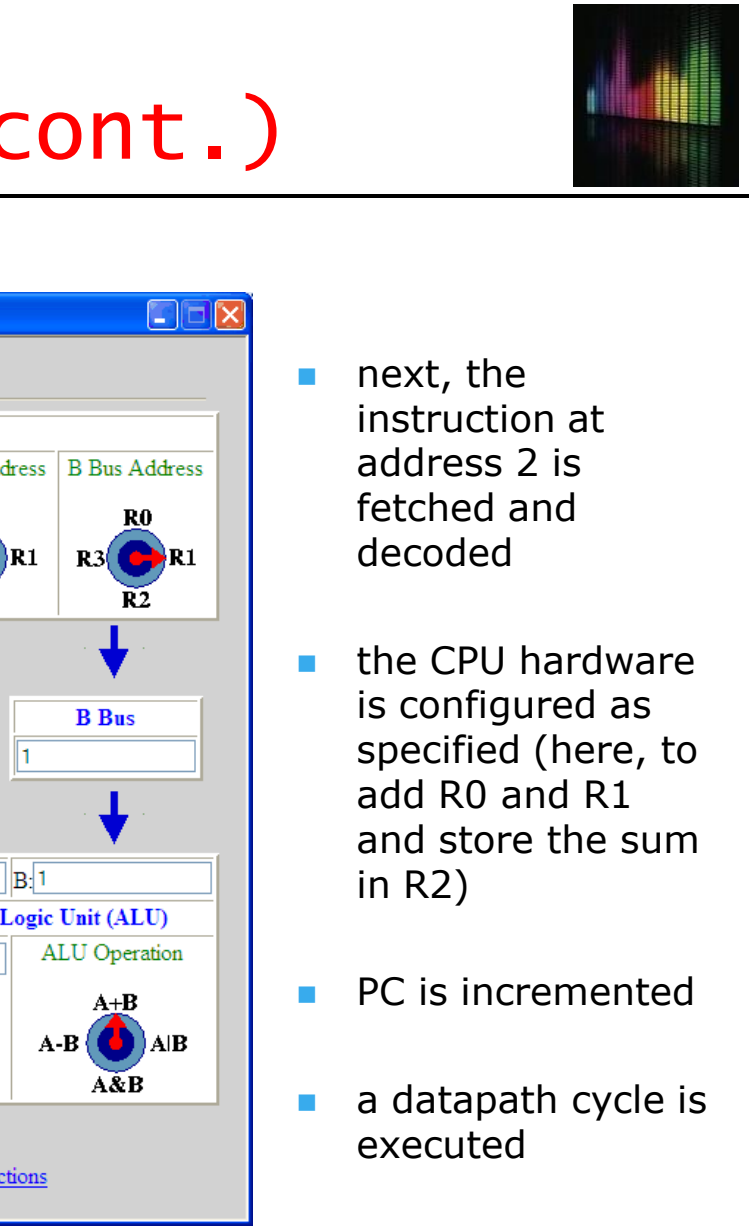
- when execution begins, the instruction at address 0 is fetched and decoded
- the CPU hardware is configured as specified (here, to copy data from address 5 to R0)
- PC is incremented
- a datapath cycle is executed

Simulator Example (cont.)



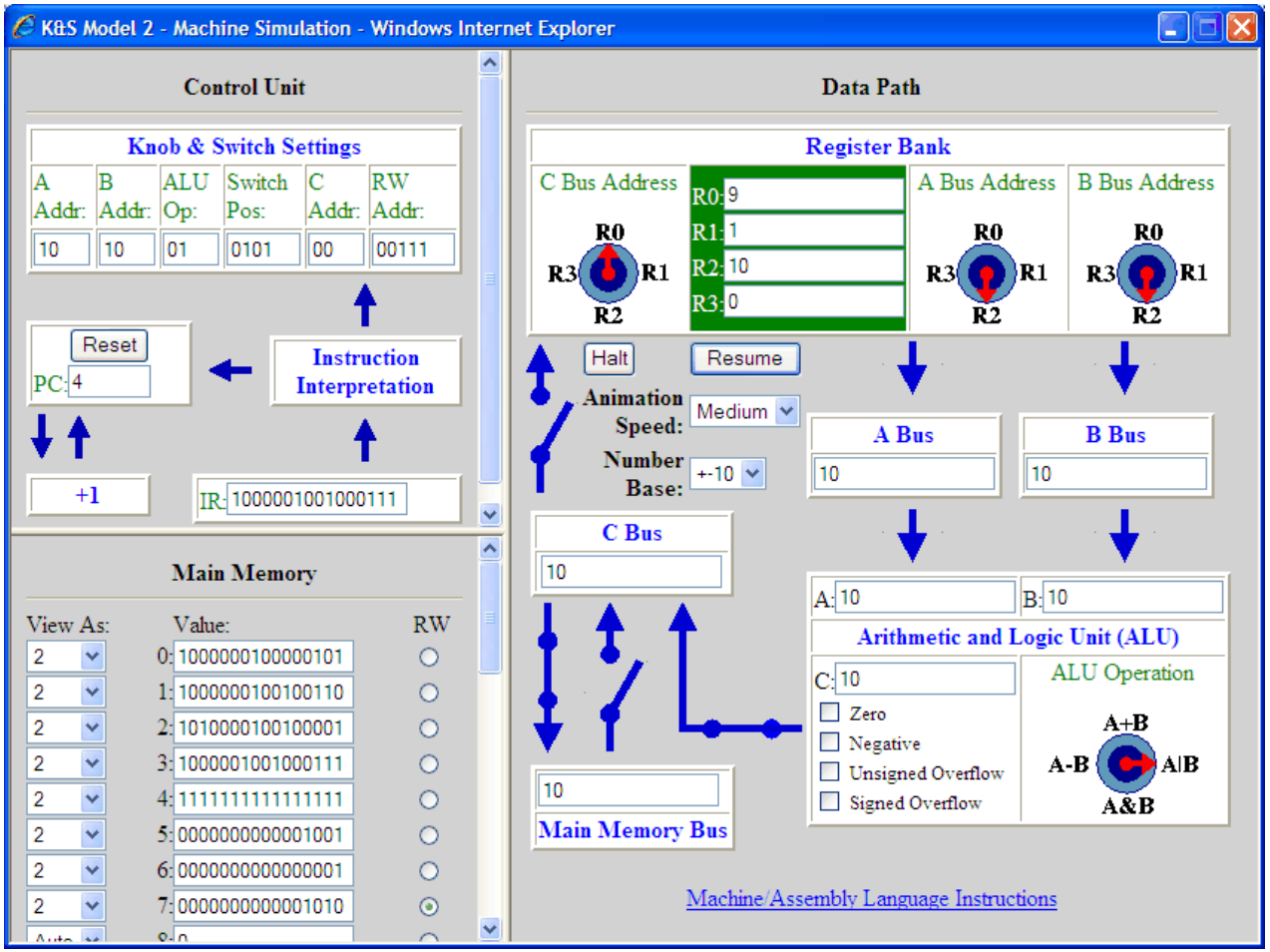
- next, the instruction at address 1 is fetched and decoded
- the CPU hardware is configured as specified (here, to copy data from address 6 to R1)
- PC is incremented
- a datapath cycle is executed

Simulator Example (cont.)



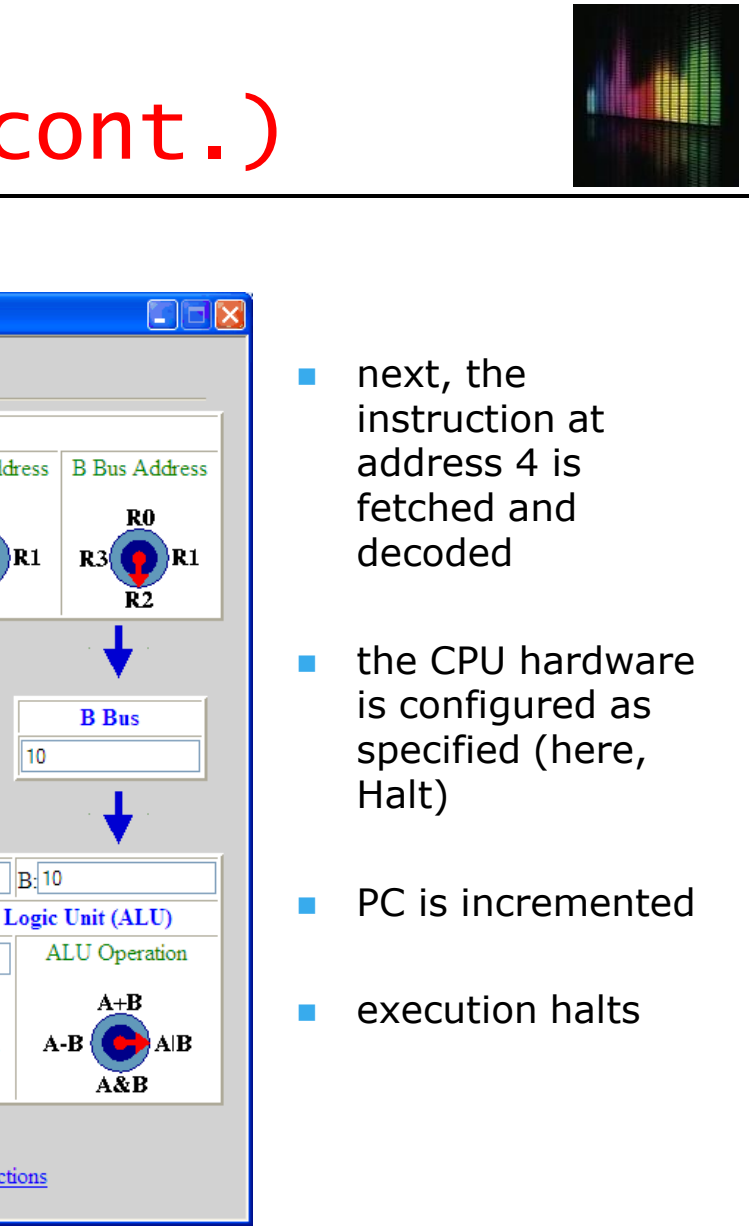
- next, the instruction at address 2 is fetched and decoded
- the CPU hardware is configured as specified (here, to add R0 and R1 and store the sum in R2)
- PC is incremented
- a datapath cycle is executed

Simulator Example (cont.)



- next, the instruction at address 3 is fetched and decoded
- the CPU hardware is configured as specified (here, the value in R2 is copied to address 7)
- PC is incremented
- a datapath cycle is executed

Simulator Example (cont.)



- next, the instruction at address 4 is fetched and decoded
- the CPU hardware is configured as specified (here, Halt)
- PC is incremented
- execution halts

The Role of I/O Devices

recall: input/output devices enable the user to interact with the computer

- input devices – keyboards, mice, scanners, …
- output devices – display screens, speakers, printers, …

the first programmable computers (1950s) and personal computers (1970s) were designed to execute one program at a time

- the user entered program instructions and data directly into main memory via input devices
- the user instructed the CPU to fetch program instructions and execute them in sequential order
- the user viewed results by sending the contents of the memory to an output device

most modern computers allow for multitasking (multiple programs can be active simultaneously)

- instructions and data associated with each program must be loaded into separate portions of main memory
- when control switches from one program to another, the CPU must save the state of the current program and locate the portion of memory for the new program
- memory partitioning and control switches are managed by the operating system

Machine vs. Assembly Language

programming in machine languages is difficult and tedious

- machine code is nearly impossible to understand and debug

assembly languages were developed in the early 1950s to simplify programming

- assembly languages allow for words to substitute for bit patterns
- also, allow for variable names instead of numerical addresses

Operation	Machine-Language Instruction	Assembly-Language Instruction
Add contents of two registers, then store result in another register e.g., R0 = R1 + R2	1010000100 RR RR RR e.g., 1010000100 00 01 10	ADD [REG] [REG] [REG] e.g., ADD R0 R1 R2
Subtract contents of two registers, then store result in another register e.g., R0 = R1 − R2	1010001000 RR RR RR e.g., 1010001000 00 01 10	SUB [REG] [REG] [REG] e.g., SUB R0 R1 R2
Load contents of memory location into register e.g., R3 = MM5	100000010 RR MMM e.g., 100000010 11 00101	LOAD [REG] [MEM] e.g., LOAD R3 5
Store contents of register into memory location e.g., MM5 = R3	100000100 RR MMMMM e.g., 100000100 11 00101	STORE [MEM] [REG] e.g., STORE 5 R3
Move contents of one register into another register e.g., R1 = R0	100100010000 RR RR e.g., 100100010000 01 00	MOVE [REG] [REG] e.g., MOVE R1 R0
Halt the machine	1111111111111111	HALT

Assembly Language Example

the stored-program computer simulator allows the user to enter instructions in assembly language

- can even toggle the view from assembly language to machine language using the "View As:" pull-down menu to the left of each memory location