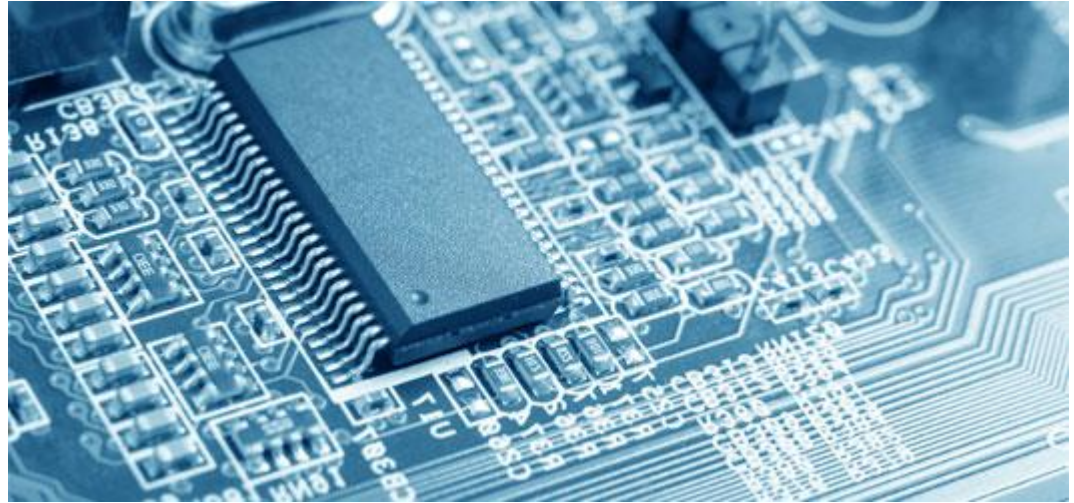




NAZARBAYEV
UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



CSCI502 – Hardware/Software Co-Design

Lecture 2 – Computer Architecture Basics: Memory. A Simple Computer. Instruction Set Architectures

14-16 January 2018

Course Logistics

Reference Reading (available in Moodle and library):

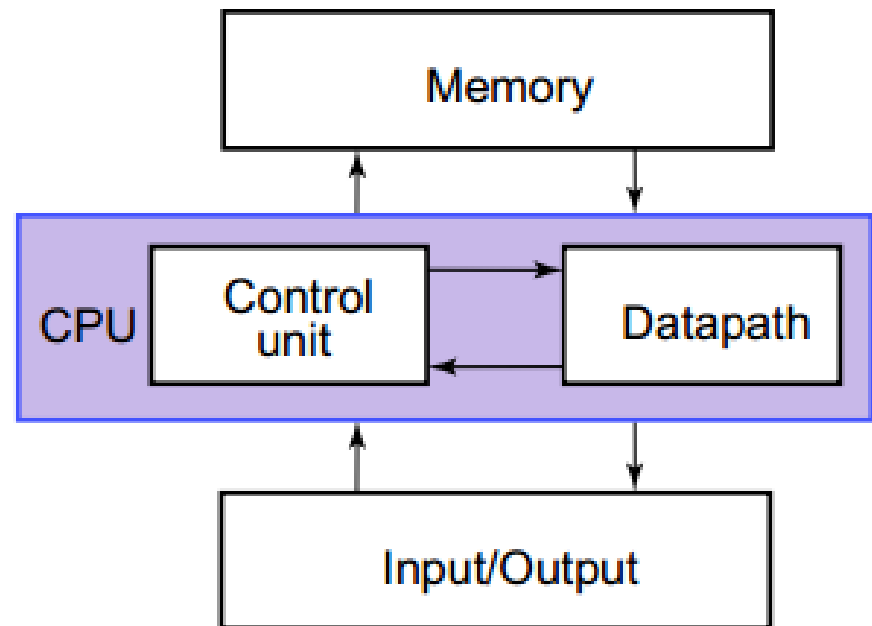
Real-Time Embedded Systems: Chapter 2

Exploring BeagleBone. 2nd edition: Chapter 5

**Logic and Computer Design Fundamentals. 5th edition:
Chapters 6 – 9 (slide relevant material)**

General Computer Architecture

- **CPU**: the “brain” of a computer
 - **Control unit** does calculations on data in **datapath**
- **Memory**: stores data (for later use)
- **Input/Output**: interface to outside (disk, network, monitor, keyboard, mouse, etc.)



Registers

- ❑ Register – a collection of binary storage elements
- ❑ More often, think of a register as storing a vector of binary values (a binary word)
- ❑ Frequently used to perform simple data storage and data movement and processing operations

Memory Definitions

- ▶ **Memory** — A collection of storage cells together with the necessary circuits to transfer information to and from them.
- ▶ **Memory Organization** — the basic architectural structure of a memory in terms of how data is accessed.
- ▶ **Random Access Memory (RAM)** — a memory organized such that data can be transferred to/or from any cell (or collection of cells) in a time that is not dependent upon the particular cell selected.
- ▶ **Memory Address** — A vector of bits that identifies a particular memory element (or collection of elements).

A d d r e s s e s	0xFFFFFFFF	1000 0000
	
	
	0x00000008	0100 1001
	0x00000007	1100 1100
	0x00000006	0110 1110
	0x00000005	0110 1110
	0x00000004	0000 0000
	0x00000003	0110 1011
	0x00000002	0101 0001
	0x00000001	1100 1001
	0x00000000	0100 1111

Main Memory

Memory Organization

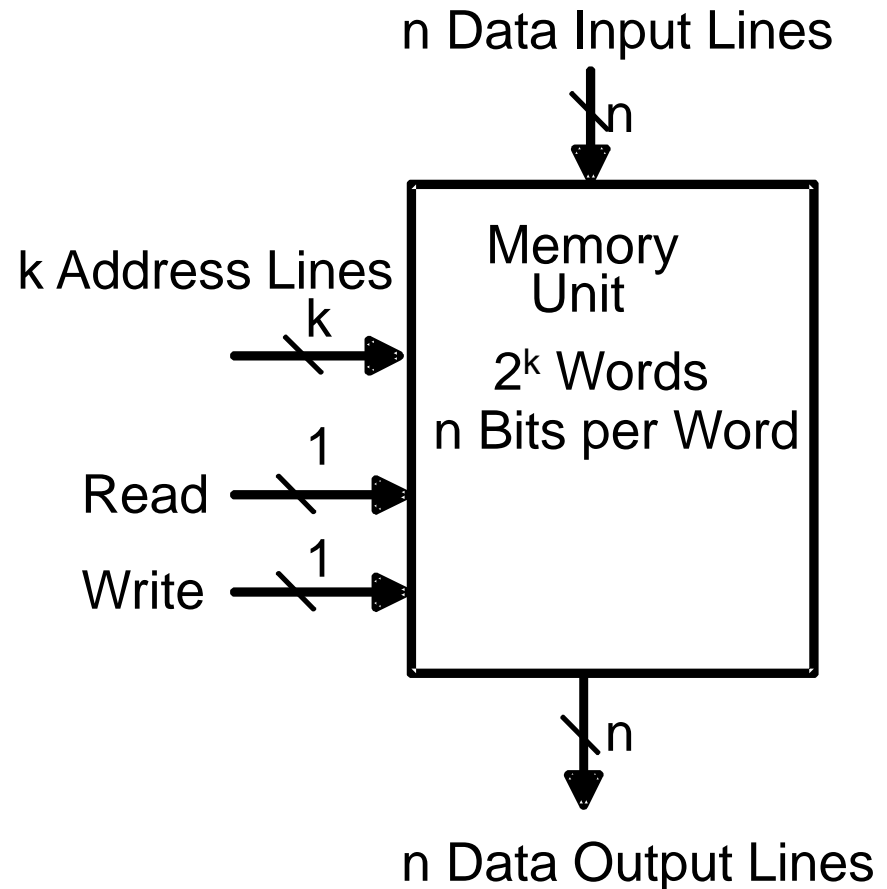
- ▶ Organized as an indexed array of words. Value of the index for each word is the memory address.
- ▶ Often organized to fit the needs of a particular computer architecture. Some historically significant computer architectures and their associated memory organization:
 - ▶ Digital Equipment Corporation PDP-8 – used a 12-bit address to address 4096 12-bit words.
 - ▶ IBM 360 – used a 24-bit address to address 16,777,216 8-bit bytes, or 4,194,304 32-bit words.
 - ▶ Intel 8080 – (8-bit predecessor to the 8086 and the current Intel processors) used a 16-bit address to address 65,536 8-bit bytes.

A d d r e s s e s	0xFFFFFFFF	1000 0000
	
	
	0x00000008	0100 1001
	0x00000007	1100 1100
	0x00000006	0110 1110
	0x00000005	0110 1110
	0x00000004	0000 0000
	0x00000003	0110 1011
	0x00000002	0101 0001
	0x00000001	1100 1001
	0x00000000	0100 1111

Main Memory

Memory Block Diagram

- ▶ A basic memory system is shown here:
- ▶ k address lines are decoded to address 2^k words of memory.
- ▶ Each word is n bits.
- ▶ Read and Write are single control lines defining the simplest of memory operations.



Memory Organization Example

- ▶ Example memory contents:
- ▶ A memory with 3 address bits & 8 data bits has:
- ▶ $k = 3$ and $n = 8$ so $2^3 = 8$ addresses labeled 0 to 7.
- ▶ $2^3 = 8$ words of 8-bit data

Memory Address		Memory Content
Binary	Decimal	
0 0 0	0	1 0 0 0 1 1 1 1
0 0 1	1	1 1 1 1 1 1 1 1
0 1 0	2	1 0 1 1 0 0 0 1
0 1 1	3	0 0 0 0 0 0 0 0
1 0 0	4	1 0 1 1 1 0 0 1
1 0 1	5	1 0 0 0 0 1 1 0
1 1 0	6	0 0 1 1 0 0 1 1
1 1 1	7	1 1 0 0 1 1 0 0

Basic Memory Operations

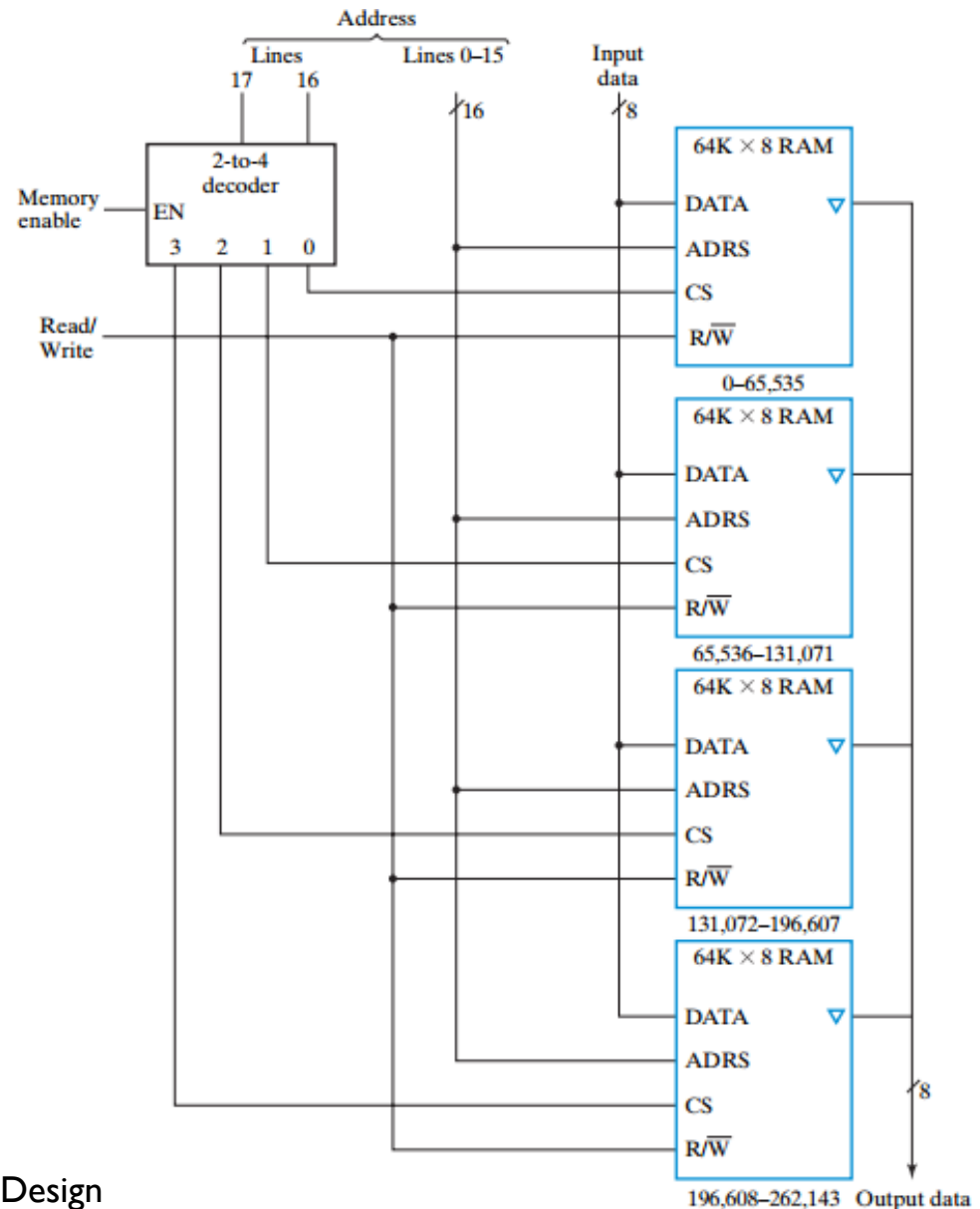
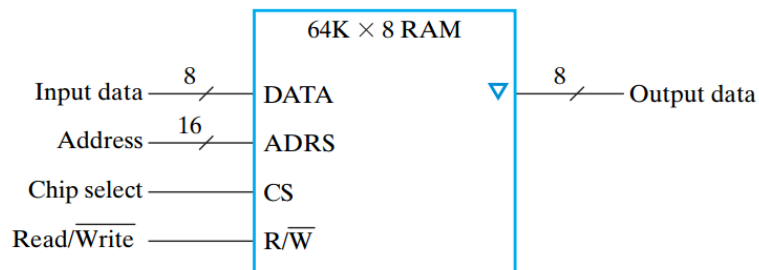
- ❖ Memory operations require the following:
 - ❑ **Data** — data written to, or read from, memory as required by the operation.
 - ❑ **Address** — specifies the memory location to operate on. The address lines carry this information into the memory. Typically: n bits specify locations of 2^n words.
 - ❑ **An operation** — Information sent to the memory and interpreted as control information which specifies the type of operation to be performed. Typical operations are READ and WRITE. Others are READ followed by WRITE and a variety of operations associated with delivering blocks of data. Operation signals may also specify timing info.

Basic Memory Operations

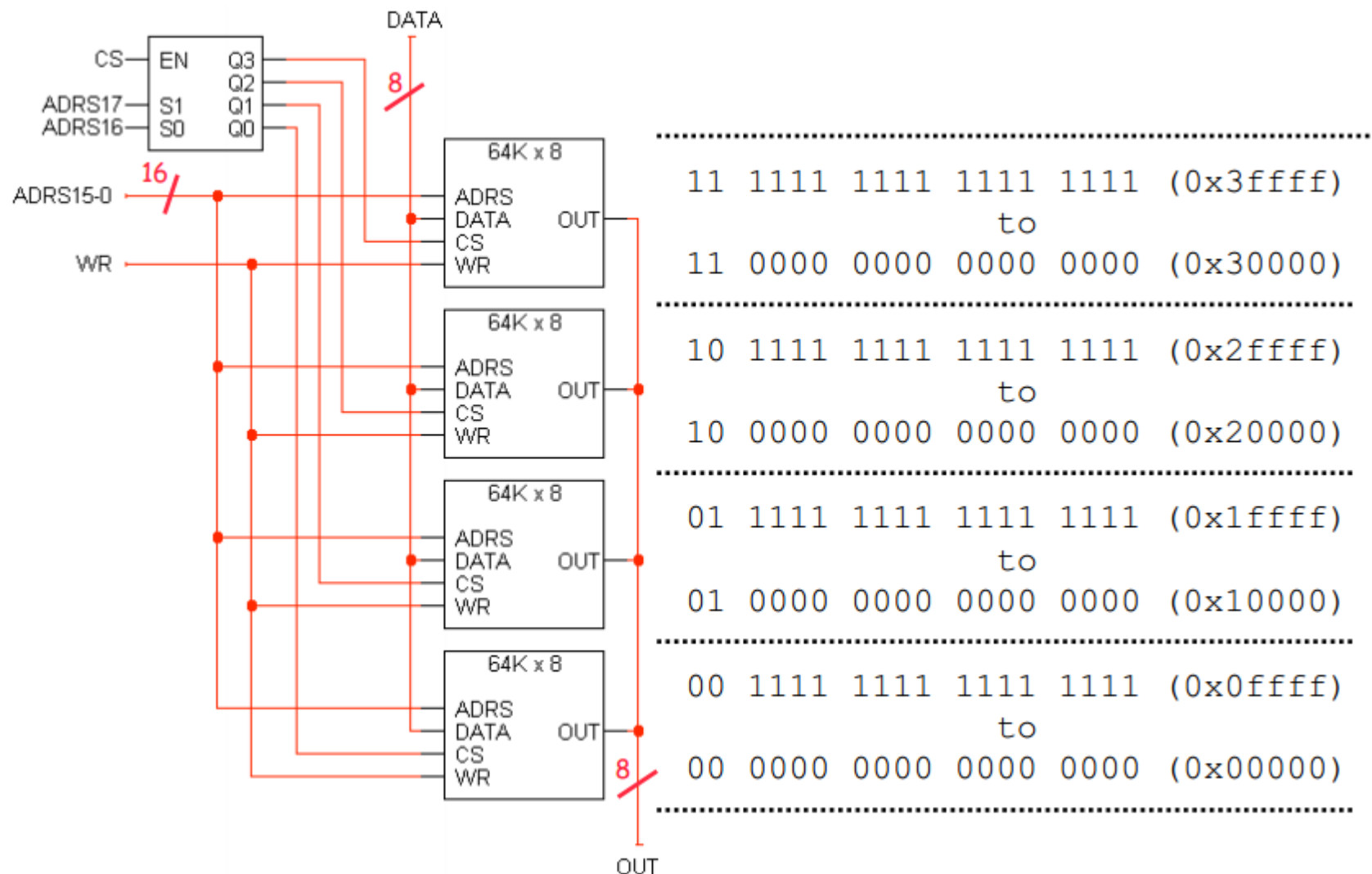
- ▶ **Read Memory** — an operation that reads a data value stored in memory:
 - ▶ Place a valid address on the address lines.
 - ▶ Wait for the read data to become stable.
- ▶ **Write Memory** — an operation that writes a data value to memory:
 - ▶ Place a valid address on the address lines and valid data on the data lines.
 - ▶ Toggle the memory write control line

Making Larger Memories

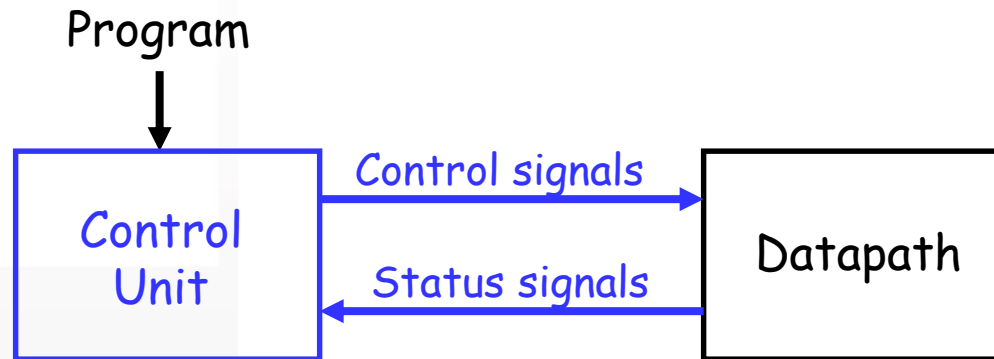
- ▶ Using the CS lines, we can make larger memories from smaller ones by tying all address, data, and R/W lines in parallel, and using the decoded higher order address bits to control CS.
- ▶ Example: using the 64K Word by 8-Bit RAM, we construct a 256K x 8 bit RAM. \Rightarrow



Address Ranges



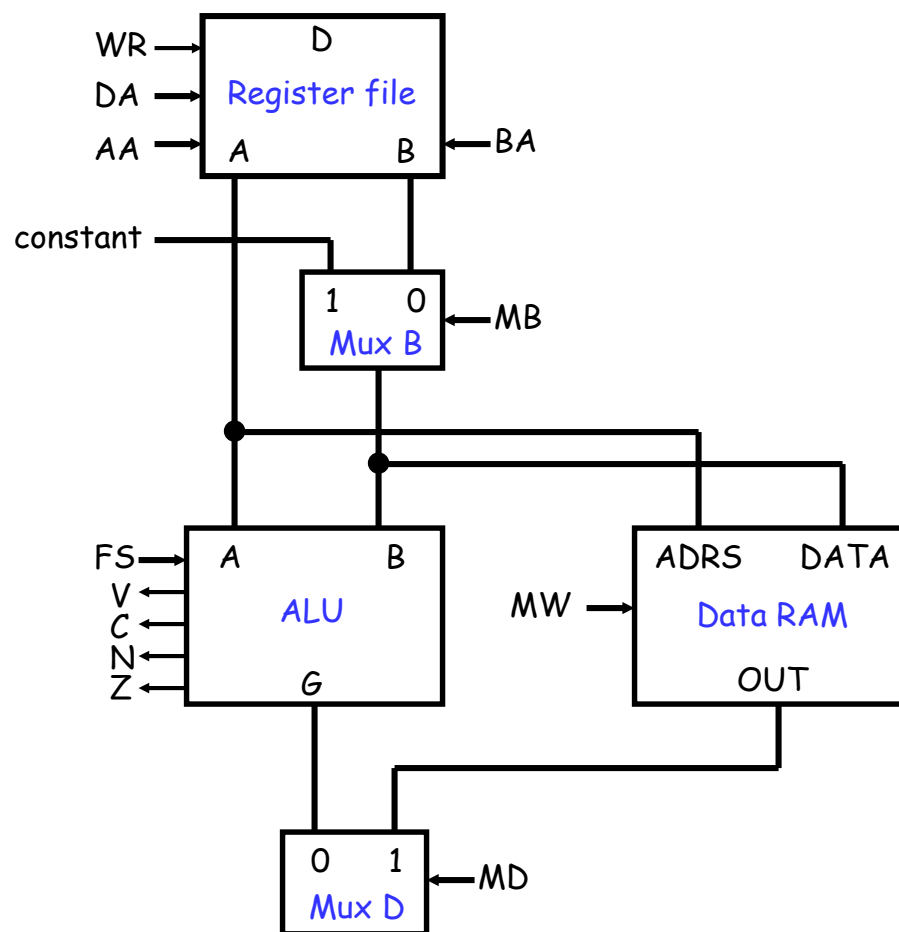
Block Diagram of a Processor



- ▶ The **control unit** connects programs with the **datapath**.
 - ▶ It converts program instructions into control words for the datapath
 - ▶ It executes program instructions in the correct sequence.
 - ▶ It generates the “constant” input for the datapath.
- ▶ The **datapath** also sends information back to the control unit. For instance, the ALU status bits can be inspected by branch instructions to alter a program’s control flow.

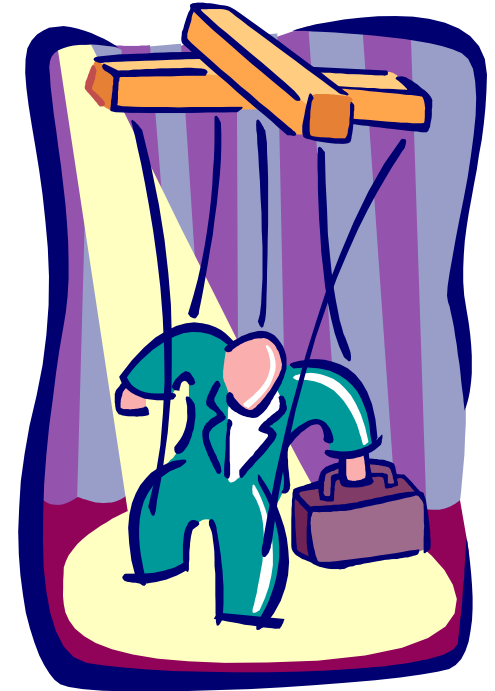
Datapath Review

- ▶ Set $WR = 1$ to write one of the registers.
- ▶ DA is the register to save to.
- ▶ AA and BA select the source registers.
- ▶ MB chooses a register or a constant operand.
- ▶ FS selects an ALU operation.
- ▶ $MW = 1$ to write to memory.
- ▶ MD selects between the ALU result and the RAM output.
- ▶ V, C, N and Z are status bits.



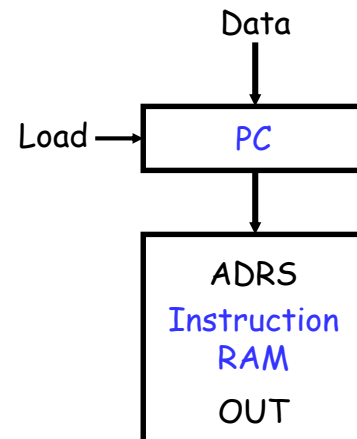
Control Units

- ▶ In real computers, the datapath actions are determined by the **program** that is loaded and running.
- ▶ A **control unit** is responsible for generating the correct control signals for a **datapath**, based on the program code.



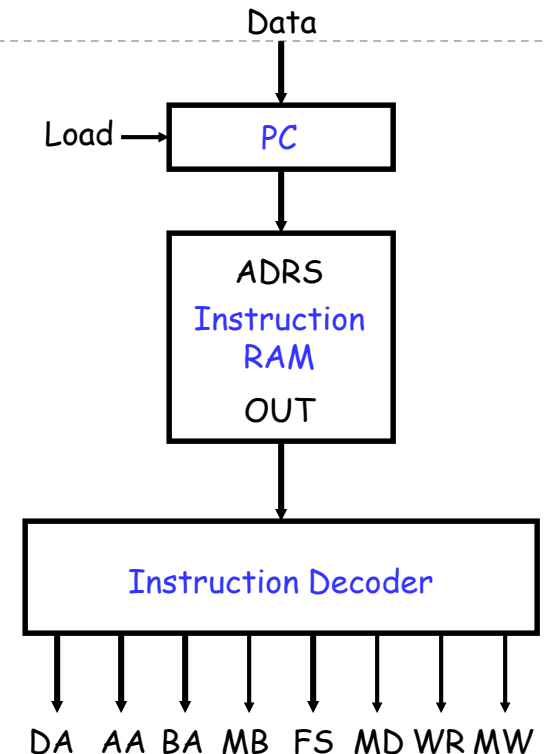
Program Counter

- ▶ A **program counter** or **PC** addresses the instruction memory, to keep track of the instruction currently being executed.
- ▶ On each clock cycle, the counter does one of two things.
 - ▶ If **Load = 0**, the PC increments, so the next instruction in memory will be executed.
 - ▶ If **Load = 1**, the PC is updated with **Data**, which represents some address specified in a jump or branch instruction.



Instruction Decoder

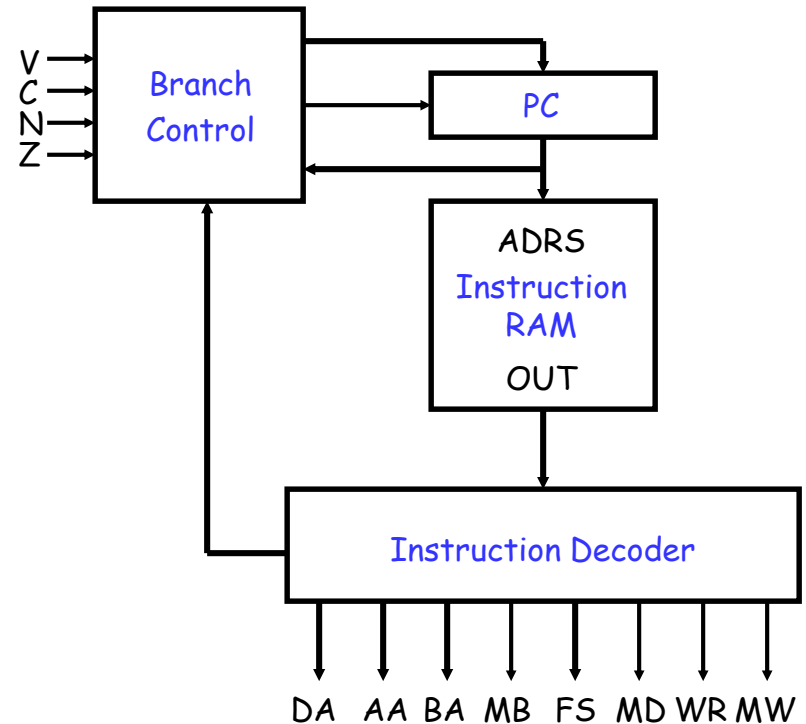
- ▶ The **instruction decoder** is a combinational circuit that takes a machine language instruction and produces the matching control signals for the datapath.
- ▶ These signals tell the datapath which registers or memory locations to access, and what ALU operations to perform.



(to the datapath)

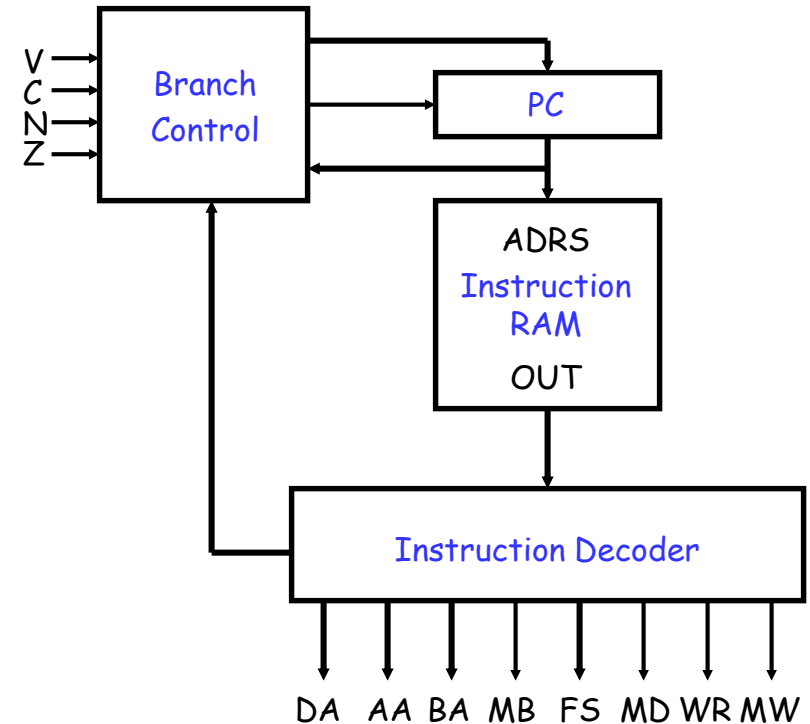
Jumps and Branches

- ▶ Finally, the **branch control unit** decides what the PC's next value should be.
 - ▶ For jumps, the PC should be loaded with the target address specified in the instruction.
 - ▶ For branch instructions, the PC should be loaded with the target address only if the corresponding status bit is true.
 - ▶ For all other instructions, the PC should just increment.

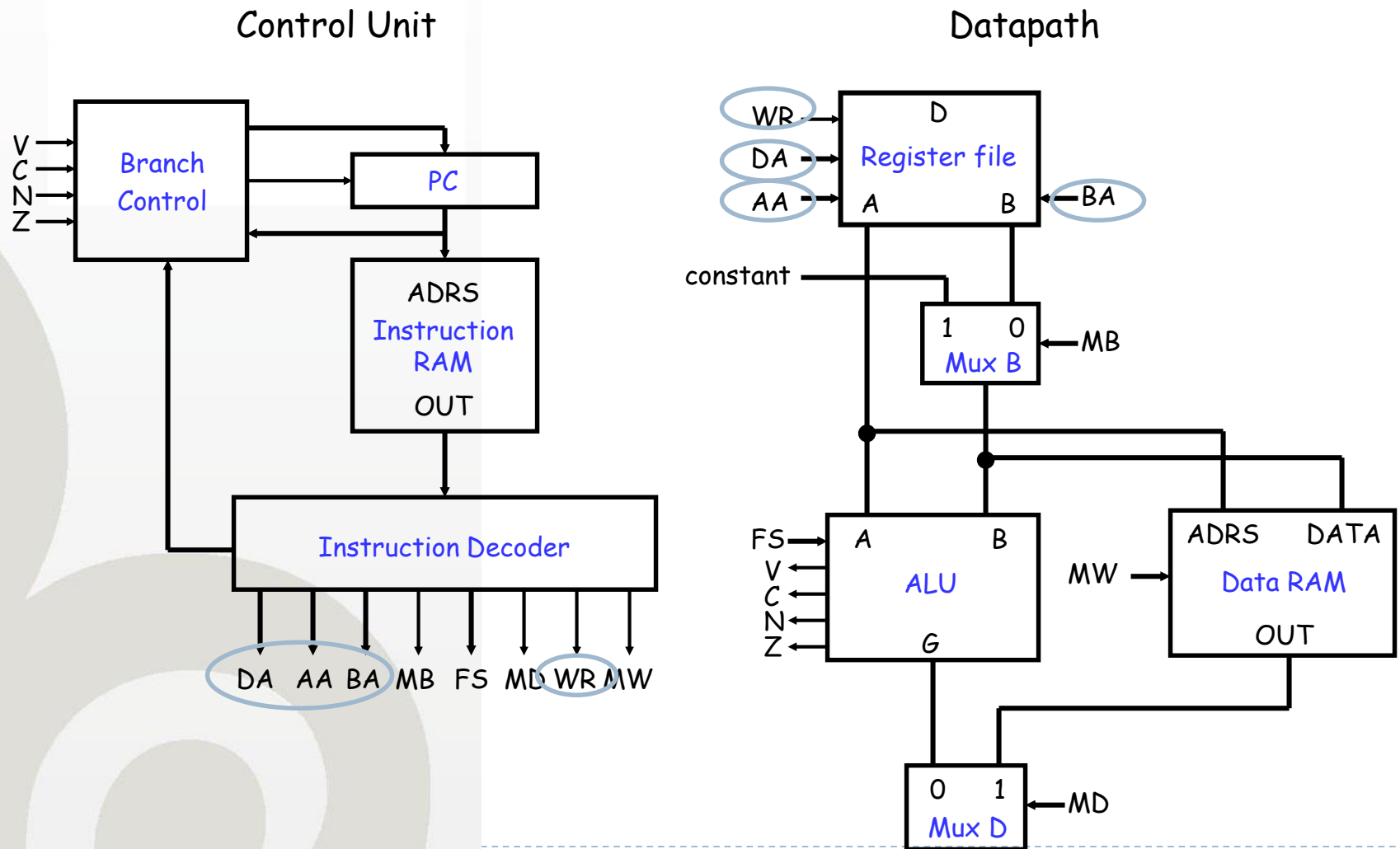


That's It!

- ▶ This is the basic control unit. On each clock cycle:
 1. An instruction is read from the instruction memory.
 2. The instruction decoder generates the matching datapath control word.
 3. Datapath registers are read and sent to the ALU or the data memory.
 4. ALU or RAM outputs are written back to the register file.
 5. The PC is incremented, or reloaded for branches and jumps.



The Whole Processor



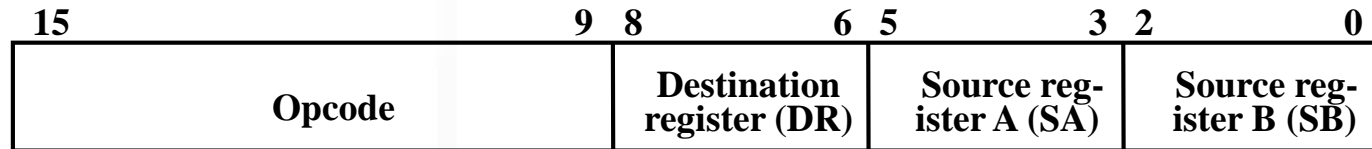
Instruction Set Architecture (ISA) for a Simple Computer

- ▶ A programmable system uses a sequence of *instructions* to control its operation
- ▶ An typical instruction specifies:
 - ▶ Operation to be performed
 - ▶ Operands to use, and
 - ▶ Where to place the result, or
 - ▶ Which instruction to execute next
- ▶ Instructions are stored in RAM or ROM as a *program*
- ▶ The addresses for instructions in a computer are provided by a ***program counter (PC)*** that can
 - ▶ Count up
 - ▶ Load a new address based on an instruction and, optionally, status information
- ▶ Executing an instruction – activating the necessary sequence of operations specified by the instruction

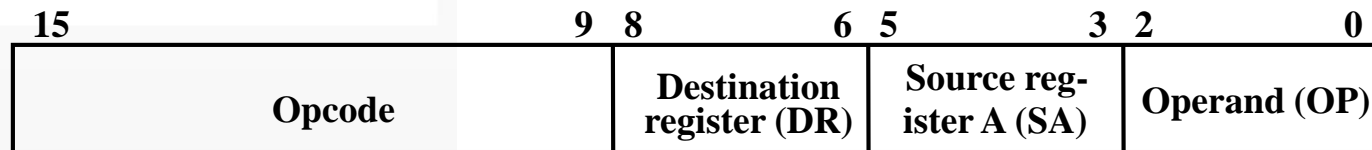
ISA: Instruction Specifications

- ▶ The specifications provide:
 - ▶ The name of the instruction
 - ▶ The instruction's opcode
 - ▶ A shorthand name for the opcode called a ***mnemonic***
 - ▶ A specification for the instruction format
 - ▶ A register transfer description of the instruction, and
 - ▶ A listing of the status bits that are meaningful during an instruction's execution

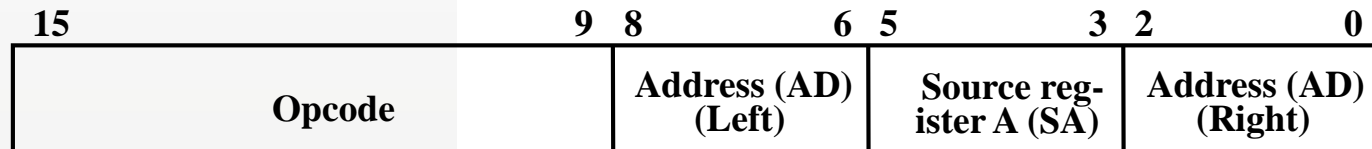
ISA: Instruction Format



(a) Register



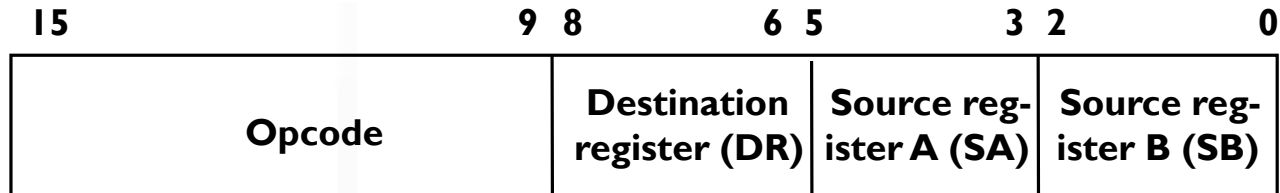
(b) Immediate



(c) Jump and Branch

- ▶ The three formats are: Register, Immediate, and Jump and Branch
- ▶ All formats contain an Opcode field in bits 9 through 15.
- ▶ The Opcode specifies the operation to be performed
- ▶ More details on each format are provided on the next three slides

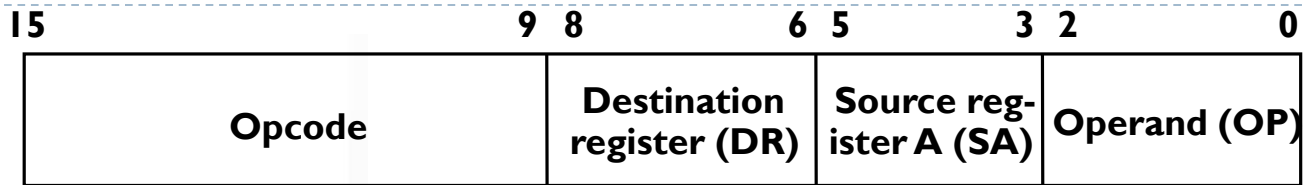
ISA: Instruction Format



(a) Register

- ▶ This format supports instructions represented by:
 - ▶ $R1 \leftarrow R2 + R3$
 - ▶ $R1 \leftarrow \text{sl } R2$
- ▶ There are three 3-bit register fields:
 - ▶ DR - specifies destination register (R1 in the examples)
 - ▶ SA - specifies the A source register (R2 in the first example)
 - ▶ SB - specifies the B source register (R3 in the first example and R2 in the second example)

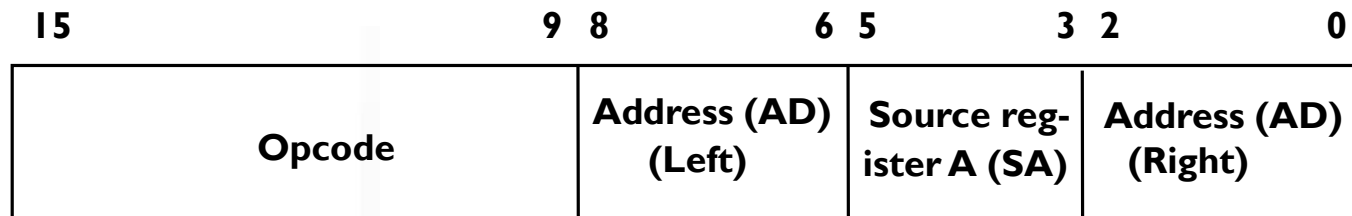
ISA: Instruction Format



(b) Immediate

- ▶ This format supports instructions described by:
 - ▶ $R1 \leftarrow R2 + 3$
- ▶ The B Source Register field is replaced by an Operand field OP which specifies a constant.
- ▶ The Operand:
 - ▶ 3-bit constant
 - ▶ Values from 0 to 7
- ▶ The constant:
 - ▶ Zero-fill (on the left of) the Operand to form N-bit constant (due to N-bit bus length), i.e. the N-bit binary word for values 0 through 7

ISA: Instruction Format



(c) Jump and Branch

- ▶ This instruction supports changes in the sequence of instruction execution by adding an extended, 6-bit, signed 2s-complement *address offset* to the PC value
- ▶ The 6-bit Address (AD) field replaces the DR and SB fields
 - ▶ Example: Suppose that a jump is specified by the Opcode and the PC contains 45 (0...0101101) and Address contains -12 (110100). Then the new PC value will be:
 $0...0101101 + (1...110100) = 0...0100001$ ($45 + (-12) = 33$)
- ▶ The SA field is retained to permit jumps and branches on N or Z based on the contents of Source register A

ISA: Instruction Specifications

Instruction	Opcode	Mne- monic	Format	Description	Status Bits
Move A	0000000	MOVA	RD, RA	$R[DR] \leftarrow R[SA]^*$	N, Z
Increment	0000001	INC	RD, RA	$R[DR] \leftarrow R[SA] + 1^*$	N, Z
Add	0000010	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]^*$	N, Z
Subtract	0000101	SUB	RD, RA, RB	$R[DR] \leftarrow R[SA] - R[SB]^*$	N, Z
Decrement	0000110	DEC	RD, RA	$R[DR] \leftarrow R[SA] - 1^*$	N, Z
AND	0001000	AND	RD, RA, RB	$R[DR] \leftarrow R[SA] \wedge R[SB]^*$	N, Z
OR	0001001	OR	RD, RA, RB	$R[DR] \leftarrow R[SA] \vee R[SB]^*$	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	$R[DR] \leftarrow R[SA] \oplus R[SB]^*$	N, Z
NOT	0001011	NOT	RD, RA	$R[DR] \leftarrow \overline{R[SA]}^*$	N, Z
Move B	0001100	MOVB	RD, RB	$R[DR] \leftarrow R[SB]^*$	

For all of these instructions, $PC \leftarrow PC + I$ is also executed to prepare for the next cycle.

ISA: Instruction Specifications

Instruction	Opcode	Mne- monic	Format	Description	Status Bits
Shift Right	0001101	SHR	RD, RB	$R[DR] \leftarrow sr\ R[SB]^*$	
Shift Left	0001110	SHL	RD, RB	$R[DR] \leftarrow sl\ R[SB]^*$	
Load Immediate	1001100	LDI	RD, OP	$R[DR] \leftarrow zf\ OP^*$	
Add Immediate	1000010	ADI	RD, RA, OP	$R[DR] \leftarrow R[SA] + zf\ OP^*$	N, Z
Load	0010000	LD	RD, RA	$R[DR] \leftarrow M[SA]^*$	
Store	0100000	ST	RA, RB	$M[SA] \leftarrow R[SB]^*$	
Branch on Zero	1100000	BRZ	RA, AD	if ($R[SA] = 0$) $PC \leftarrow PC + se\ AD$, N, Z if ($R[SA] \neq 0$) $PC \leftarrow PC + 1$	
Branch on Negative	1100001	BRN	RA, AD	if ($R[SA] < 0$) $PC \leftarrow PC + se\ AD$, N, Z if ($R[SA] \geq 0$) $PC \leftarrow PC + 1$	
Jump	1110000	JMP	RA	$PC \leftarrow R[SA]$	

For all of these instructions, $PC \leftarrow PC + I$ is also executed to prepare for the next cycle.

ISA: Example Instructions and Data in Memory

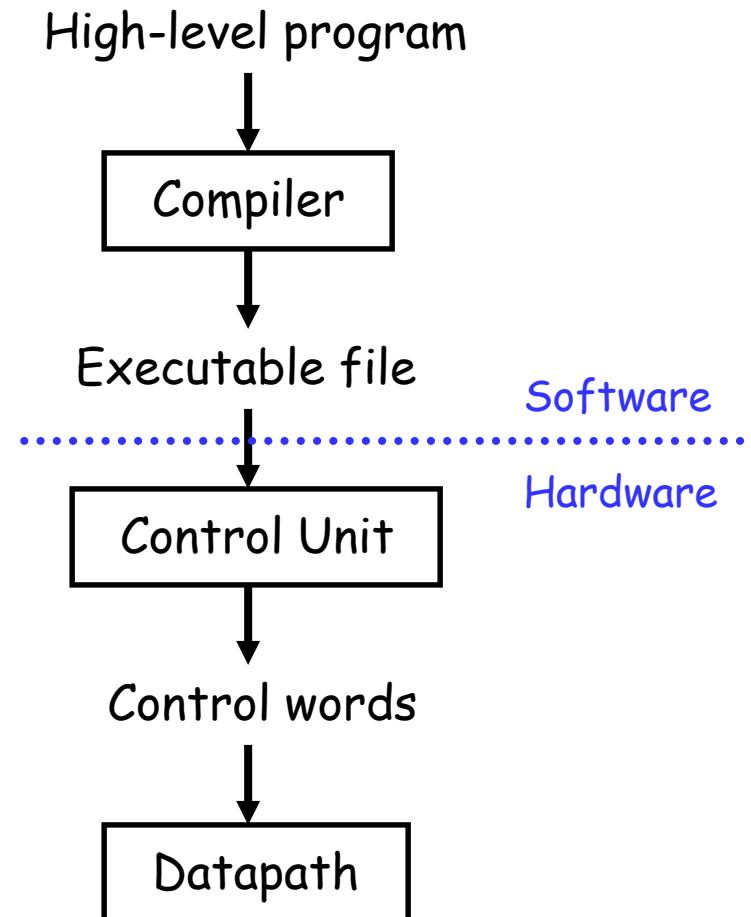
Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	$R1 \leftarrow R2 - R3$
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	$R4 = 70, R5 = 80$ $M[R4] \leftarrow R5$
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	$R2 \leftarrow R7 + 3$
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If $R6 = 0$, $PC \leftarrow PC - 20$
70	00000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

Single-Cycle Hardwired Control

- ▶ Based on the ISA defined, design a computer architecture to support the ISA
- ▶ The architecture is to fetch and execute each instruction in a single clock cycle
- ▶ A similar datapath to the one covered in the previous lecture will be used
- ▶ The control unit will be defined as a part of the design

Programming and CPUs

- ▶ Programs written in a high-level language like C, C++ must be **compiled** to produce an executable program.
- ▶ The result is a CPU-specific **machine language** program. This can be loaded into memory and executed by the processor.
- ▶ So far we focused on the part below the dotted blue line, but machine language serves as the **interface** between hardware and software.



High-Level Languages

- ▶ **High-level languages** provide many useful programming constructs.
 - ▶ For, while, and do loops
 - ▶ If-then-else statements
 - ▶ Functions and procedures for code abstraction
 - ▶ Variables and arrays for storage
- ▶ Many languages provide safety features as well.
 - ▶ Static and dynamic typechecking
 - ▶ Garbage collection
- ▶ High-level languages are also relatively portable. Theoretically, you can write one program and compile it on many different processors.
- ▶ It may be hard to understand what's so “high-level” here, until you compare these languages with...

Low-level languages

- ▶ Each CPU has its own low-level **instruction set**, or machine language, which closely reflects the CPU's design.
- ▶ Unfortunately, this means instruction sets are not easy for humans to work with!
 - ▶ Control flow is limited to “jump” and “branch” instructions, which you must use to make your own loops and conditionals.
 - ▶ Support for functions and procedures may be limited.
 - ▶ Memory addresses must be explicitly specified. You can't just declare new variables and use them!
 - ▶ Very little error checking is provided.
 - ▶ It's difficult to convert machine language programs to different processors.

Compiling

- ▶ Processors can't execute programs written in high-level languages directly, so a special program called a **compiler** is needed to translate high-level programs into low-level machine code.
- ▶ Earlier, people often wrote machine language programs by hand to make their programs faster, smaller, or both.
- ▶ Now, compilers almost always do a better job than people.
 - ▶ Programs are becoming more complex, and it's hard for humans to write and maintain large, efficient machine language code.
 - ▶ CPUs are becoming more complex. It's difficult to write code that takes full advantage of a processor's features.

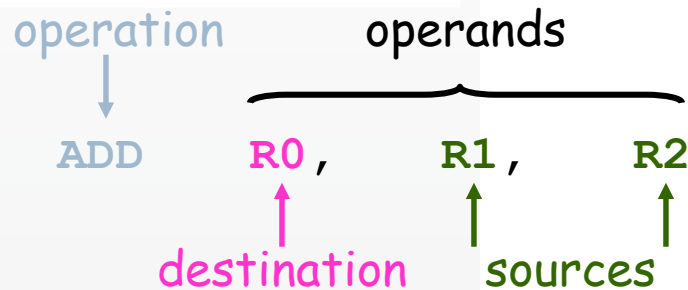
Assembly and Machine Languages

- ▶ Machine language instructions are sequences of bits in a specific order.
- ▶ To make things simpler, people typically use **assembly language**.
 - ▶ We assign “mnemonic” names to operations and operands.
 - ▶ There is (almost) a one-to-one correspondence between these mnemonics and machine instructions, so it is very easy to convert assembly programs to machine language.



Data Manipulation Instructions

- ▶ **Data manipulation** instructions correspond to ALU operations.
- ▶ For example, here is a possible addition instruction, and its equivalent using our register transfer notation:



Register transfer instruction:

$$R0 \leftarrow R1 + R2$$

- ▶ This is similar to a high-level programming statement like

$$R0 = R1 + R2$$

- ▶ Here, all of the operands are registers.

More Data Manipulation Instructions

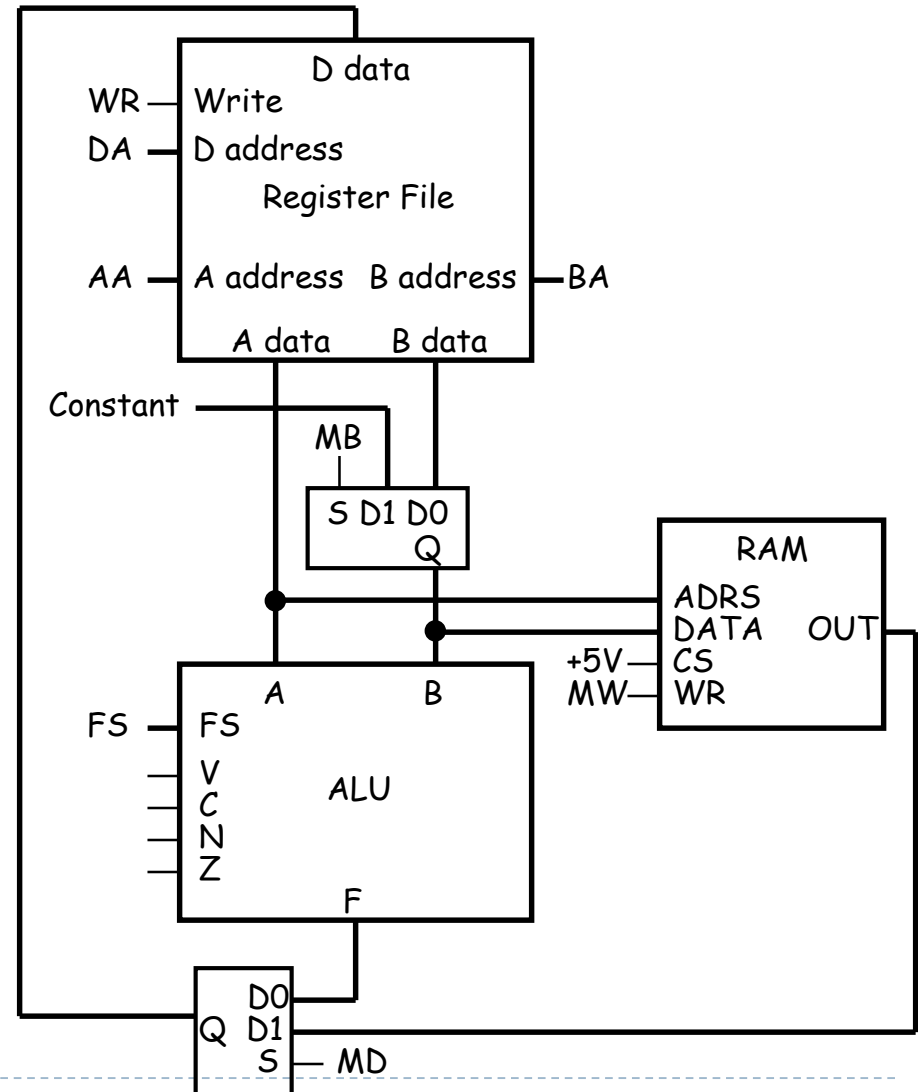
- ▶ Here are some other kinds of data manipulation instructions.

NOT	R0, R1	$R0 \leftarrow R1'$
ADD	R3, R3, #1	$R3 \leftarrow R3 + 1$
SUB	R1, R2, #5	$R1 \leftarrow R2 - 5$

- ▶ Some instructions, like the NOT, have only one operand.
- ▶ In addition to register operands, constant operands like 1 and 5 are also possible. Constants are denoted with a hash mark in front.

What about RAM?

- ▶ Recall that our ALU has direct access only to the register file.
- ▶ RAM contents must be copied to the registers before they can be used as ALU operands.
- ▶ Similarly, ALU results must go through the registers before they can be stored into memory.
- ▶ We rely on **data movement** instructions to transfer data between the RAM and the register file.

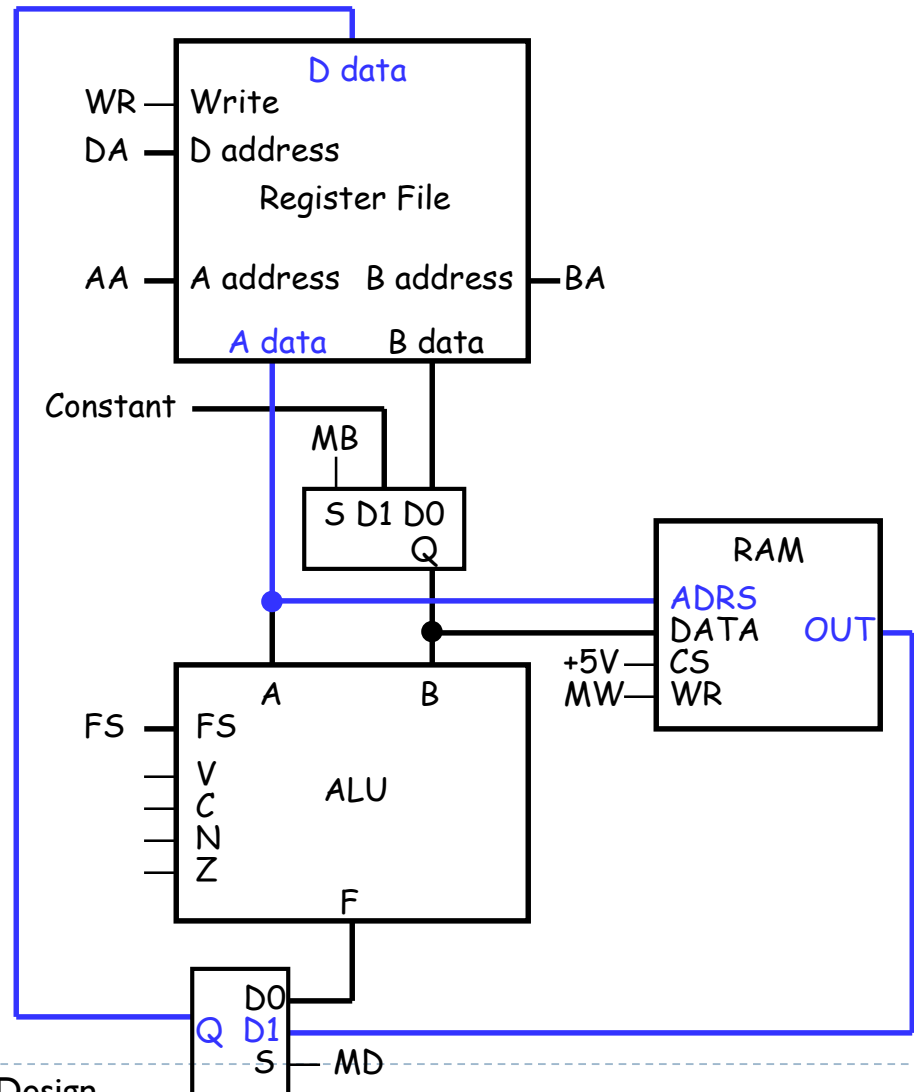


Loading a Register from RAM

- ▶ A **load** instruction copies data *from* a RAM address *to* one of the registers.

LD R1, (R3) $R1 \leftarrow M[R3]$

- ▶ Remember in our datapath, the RAM address must come from one of the registers - in the example above, R3.
- ▶ The parentheses help show which register operand holds the memory address.

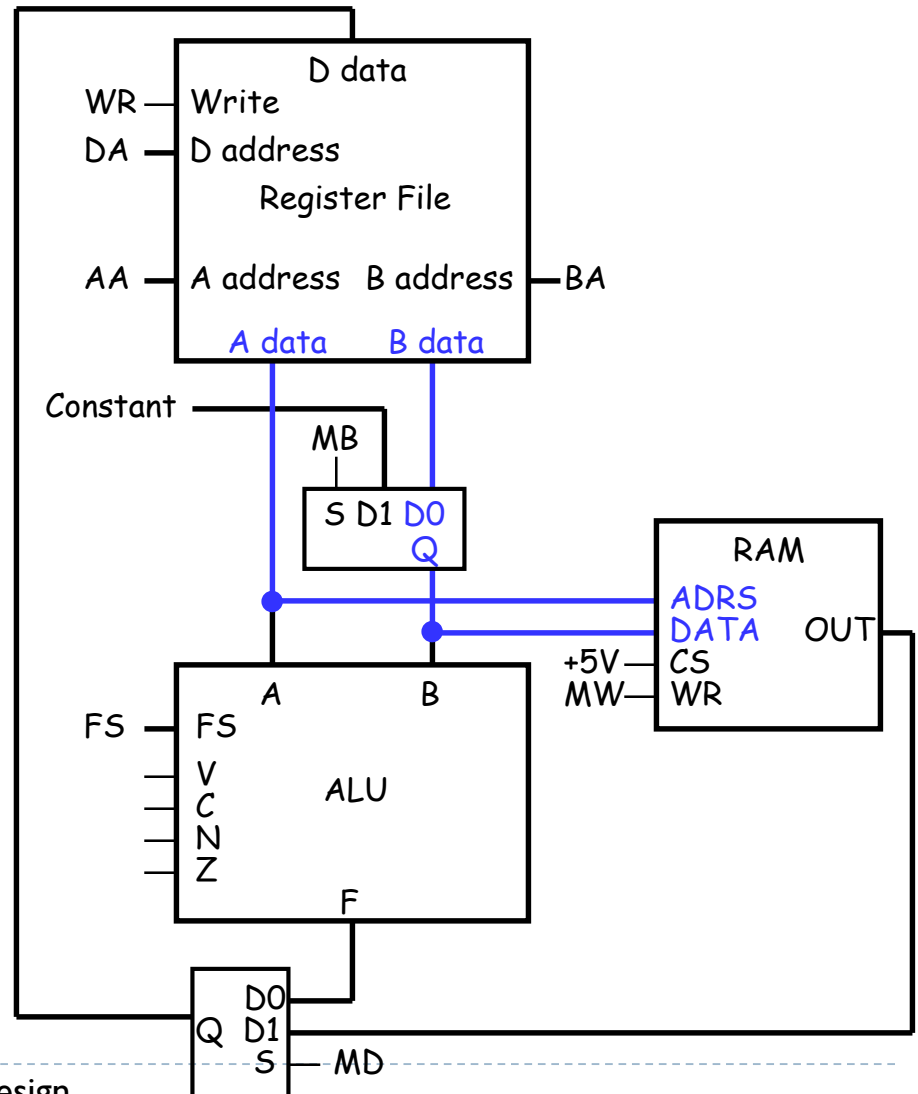


Storing a Register to RAM

- ▶ A **store** instruction copies data *from* a register *to* an address in RAM.

`ST (R3) ,R1 M[R3] ← R1`

- ▶ One register specifies the RAM address to write to - in the example above, R3.
- ▶ The other operand specifies the actual data to be stored into RAM - R1 above.

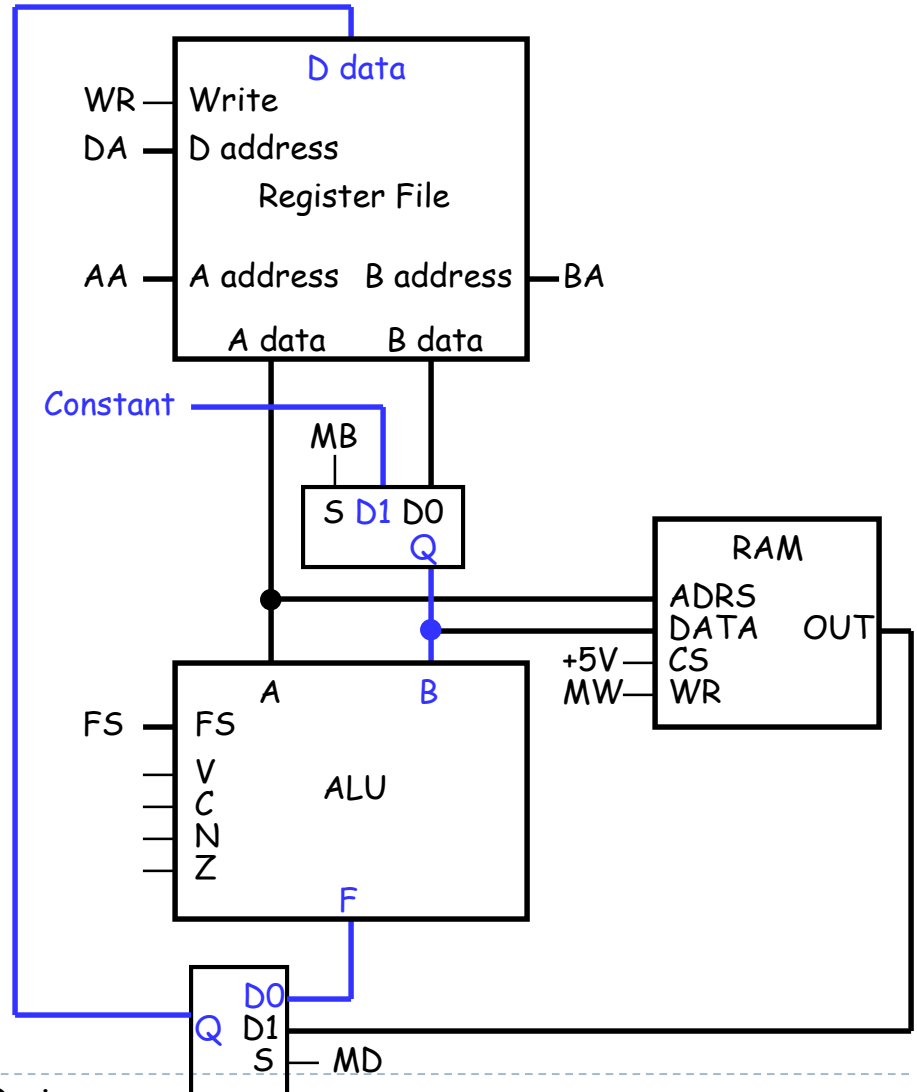


Loading a Register with a Constant

- ▶ With our datapath, it's also possible to load a constant into the register file:

`LD R1, #0` $R1 \leftarrow 0$

- ▶ Our example ALU has a “transfer B” operation (FS=10000) which lets us pass a constant up to the register file.
- ▶ This gives us an easy way to initialize registers.

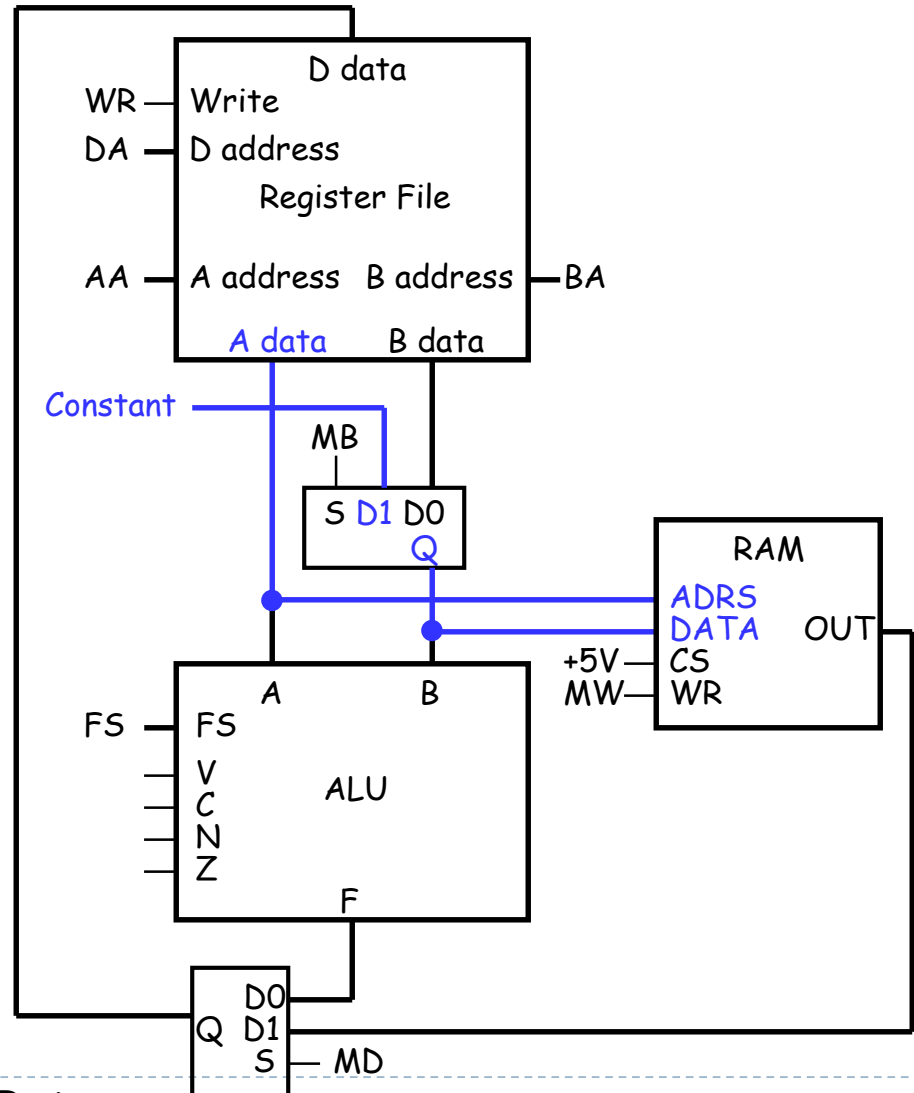


Storing a Constant to RAM

- ▶ And you can store a constant value directly to RAM too:

```
ST (R3), #0      M[R3] ← 0
```

- ▶ This provides an easy way to initialize memory contents.



The # and () are Important!

- ▶ We've seen several statements containing the # or () symbols. These are ways of specifying different **addressing modes**.
- ▶ The addressing mode we use determines which data are actually used as operands:

```
LD  R0, #1000           // R0 ← 1000
LD  R0, 1000            // R0 ← M[1000]
```

```
LD  R3, R0              // R3 ← R0
LD  R3, (R0)            // R3 ← M[R0]
```

- ▶ The design of our datapath determines which addressing modes we can use.
 - ▶ The second example above wouldn't work in our datapath. Why not?

A Small Example

- ▶ Here's an example register-transfer operation.

$$M[1000] \leftarrow M[1000] + 1$$

- ▶ This is the assembly-language equivalent:

```
LD  R0, #1000      // R0 ← 1000
LD  R3, (R0)        // R3 ← M[1000]
ADD R3, R3, #1      // R3 ← R3 + 1
ST  (R0), R3        // M[1000] ← R3
```

- ▶ An awful lot of assembly instructions are needed!
 - ▶ For instance, we have to load the memory address 1000 into a register first, and then use that register to access the RAM.
 - ▶ This is due to our relatively simple datapath design, which only allows register and constant operands to the ALU.

Fetch and Execute Cycle

- ▶ Programs are a sequence of macroinstructions or macrocode stored in the main memory in binary form.
- ▶ Macroinstructions are sequentially fetched from the main memory location pointed to by the program counter, and placed in the instruction register.
- ▶ Each instruction consists of an operation code or opcode field and zero or more operand fields.
- ▶ The control unit decodes the instruction.
- ▶ After executing the instruction, the next macroinstruction is retrieved from main memory and executed.
- ▶ Certain macroinstructions or external conditions may cause a nonconsecutive macroinstruction to be executed.
- ▶ This process is called the fetch-execute cycle (or fetch-decode-execute).
- ▶ Even when “idling,” the computer is fetching and executing an instruction that causes no effective change to the state of the CPU and is called a no-op (for no-operation).

Control Flow Instructions

- ▶ Programs consist of a lot of sequential instructions, which are meant to be executed one after another.
- ▶ Thus, programs are stored in memory so that:
 - ▶ Each program instruction occupies one address.

```
768:    LD    R0, #1000    // R0 ← 1000
769:    LD    R3, (R0)     // R3 ← M[1000]
770:    ADD   R3, R3, #1    // R3 ← R3 + 1
771:    ST    (R0), R3     // M[1000] ← R3
```

- ▶ A **program counter** (PC) keeps track of the current instruction address.
 - ▶ Ordinarily, the PC just increments after executing each instruction.
 - ▶ But sometimes we need to change this normal sequential behavior, with special control flow instructions.

Jumps

- ▶ A **jump** instruction *always* changes the value of the PC.
 - ▶ The operand specifies exactly how to change the PC.
 - ▶ For simplicity, we often use **labels** to denote actual addresses.
- ▶ For example, a program can skip certain instructions.

```
LD  R1, #10
LD  R2, #3
JMP L
K  LD  R1, #20      // These two instructions
LD  R2, #4         // would be skipped
L  ADD R3, R3, R2
ST  (R1), R3
```

- ▶ You can also use jumps to repeat instructions.

```
LD  R1, #0
F  ADD R1, R1, #1
JMP F              // An infinite loop!
```

Branches

- ▶ A **branch** instruction *may* change the PC, depending on whether a given condition is true.

```
LD  R1, #10
LD  R2, #3
BZ  R4, L      // Jump to L if R4 == 0
K   LD  R1, #20  // These instructions may be
   LD  R2, #4    //   skipped, depending on R4
L   ADD  R3, R3, R2
   ST   (R1), R3
```


High-level Control Flow

- ▶ These jumps and branches are much simpler than the control flow constructs provided by high-level languages.
- ▶ **Conditional statements** execute only if some Boolean value is true.

```
// Find the absolute value of *X
R1 = *X;
if (R1 < 0)
    R1 = -R1;                // This might not be executed
R3 = R1 + R1;
```


```
// Sum the integers from 1 to 5
R1 = 0;
for (R2 = 1; R2 <= 5; R2++)
    R1 = R1 + R2;            // This is executed five times
R3 = R1 + R1;
```

- ▶ **Loops** cause some statements to be executed many times

Translating the C If-Then Statement

- ▶ We can use branch instructions to translate high-level conditional statements into assembly code.

```
R1 = *X;  
if (R1 < 0)  
    R1 = -R1;  
R3 = R1 + R1;
```



```
LD  R1, (X)           // R1 = *X  
BNN R1, L             // Skip MUL if R1 is not negative  
MUL R1, R1, #-1       // R1 = -R1  
L   ADD R3, R1, R1     // R3 = R1 + R1
```

- ▶ Sometimes it's easier to *invert* the original condition. Here, we effectively changed the `R1 < 0` test into `R1 >= 0`.

Translating the C For loop

- ▶ Here is a translation of the for loop, using a hypothetical BGT branch.

```
R1 = 0;  
for (R2 = 1; R2 <= 5; R2++)  
    R1 = R1 + R2;  
R3 = R1 + R1;
```



```
LD    R1, #0           // R1 = 0  
LD    R2, #1           // R2 = 1  
FOR   BGT R2, #5, L     // Stop when R2 > 5  
      ADD R1, R1, R2    // R1 = R1 + R2  
      ADD R2, R2, #1    // R2++  
      JMP FOR          // Go back to the loop test  
L     ADD R3, R1, R1    // R3 = R1 + R1
```

Summary

- ▶ Machine language is the interface between software and processors.
- ▶ High-level programs must be translated into machine language before they can be run.
- ▶ There are three main categories of instructions.
 - ▶ Data manipulation operations, such as adding or shifting
 - ▶ Data transfer operations to copy data between registers and RAM
 - ▶ Control flow instructions to change the execution order
- ▶ Instruction set architectures depend highly on the host CPU's design.

Any Questions?

