

- *Real-Time operation* is a typical constraint on an embedded system /software. Concept based on meeting predetermined timing constraints rather than just executing as fast as possible.
- *Soft Real-Time*: not meeting timing constraints represents *degraded performance*
- *Hard Real-Time*: not meeting timing constraints represents *failure*
- *Firm Real-Time*: has a mixture of soft and hard constraints for various tasks
- Many *real-time systems* require definition, treatment/management, and management of formal *tasks* and *processes* (will formalize these later in the course)
- Need to develop methods for *task cooperation* to work together in a coordinated fashion and *task communication* to share data
- *Polling* or *event-driven* schemes can define the management and execution of processes and define how to interface with the external world

We will learn the role of operating systems to facilitate these

Computer System Vocabulary:

- We move signals into, out of, or throughout the system on paths called busses. In their most common implementation, busses are simply collections of wires that are carrying related electrical signals from one place to another
- The width of a bus is the number of signals or bits that it can carry simultaneously

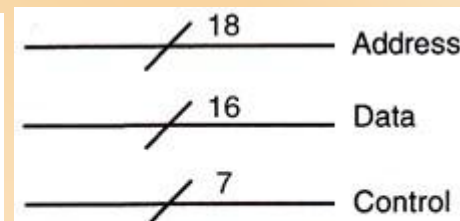
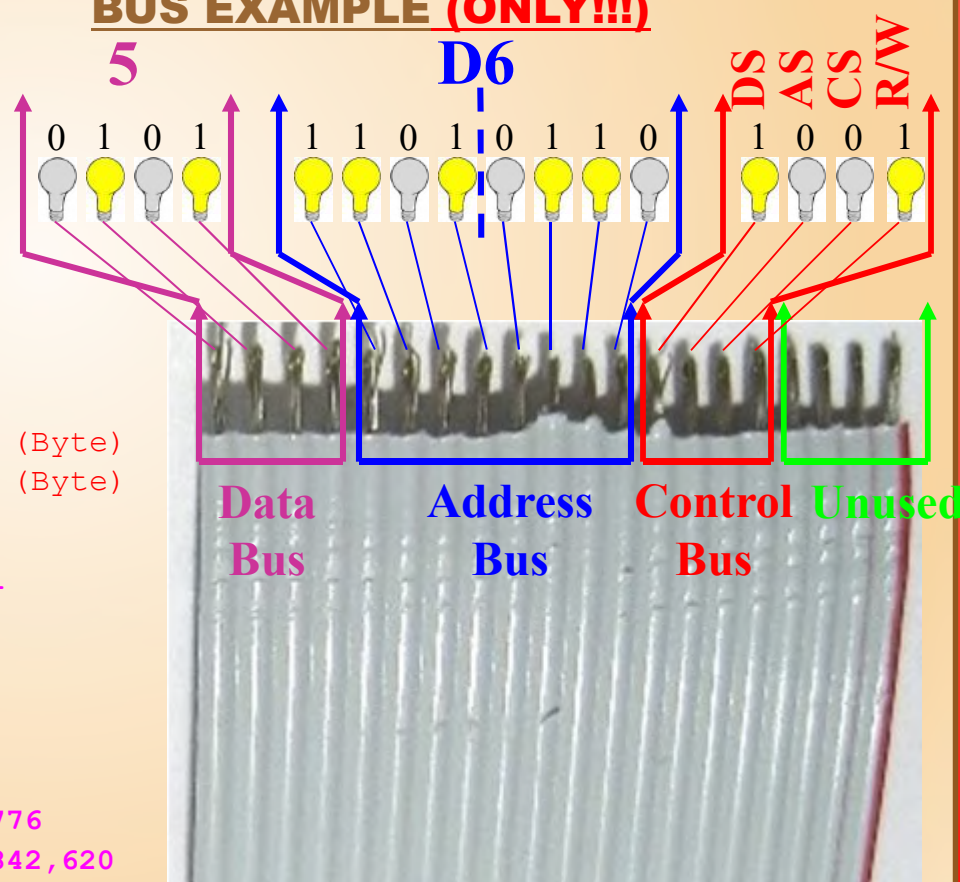


Figure 1.4 Identifying the Number of Signals in a Bus

BUS EXAMPLE (ONLY!!!)

Dec	Hex	MSB	LSB
34,231	= 85B7	1000 0101	1011 0111
		8 5	B 7
0	0000	Nibble = 4 Binary Digits	
1	0001	Byte = 8 Binary Digits	
2	0010		
3	0011		
4	0100		
5	0101	MSB = Most Significant Bit (Byte)	
6	0110	LSB = Least Significant Bit (Byte)	
7	0111	Bit = Single binary digit	
8	1000	bit	2^0 Value of 0 or 1
9	1001	Nibble	2^4 4 bits
A	1010	Byte	2^8 8 bits
B	1011	Kilobit	2^{10} 1,024
C	1100	Megabit	2^{20} 1,048,576
D	1101	Gigabit	2^{30} 1,073,741,824
E	1110	Terrabit	2^{40} 1,099,511,627,776
F	1111	Petabit	2^{50} 1,125,899,906,842,620



Computer System Vocabulary:

- The data are the key signals that are being moved around; the address signals identify where the data is coming from and where it is going to; and the control signals specify and coordinate how the data is transported.
 - Think of the arrangement as being similar to your telephone. The number you dial is the address of where your conversation will be directed, and the ring is one of the control signals indicating to the person you are calling that a call is coming in. Your voice or text message is the data that you are moving from your location to the person on the other telephone. As with your telephone, the medium carrying the signal may take many forms: copper wire, fiber-optic cable, or electromagnetic waves

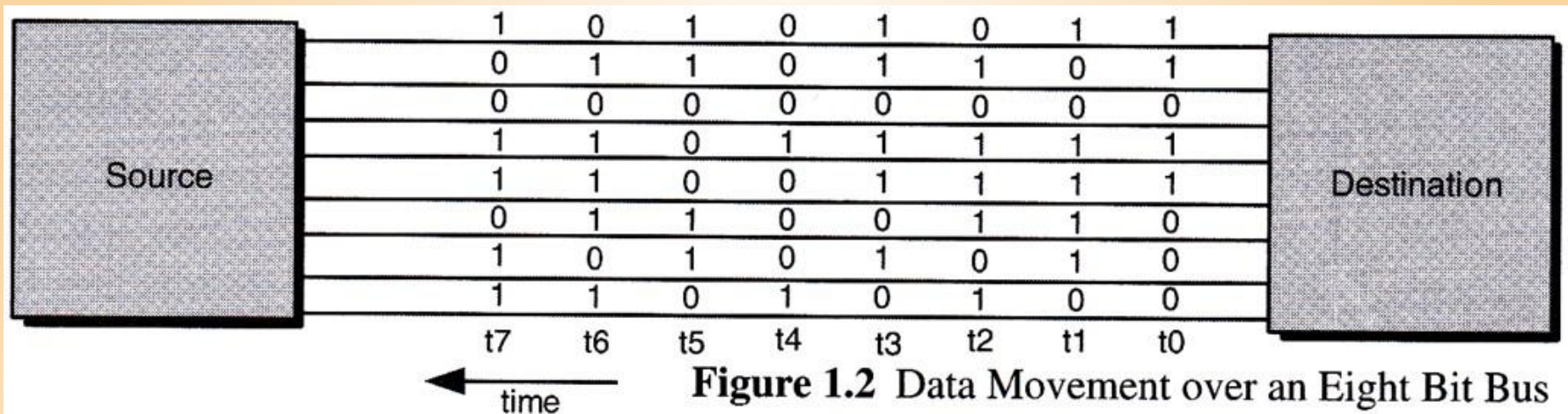
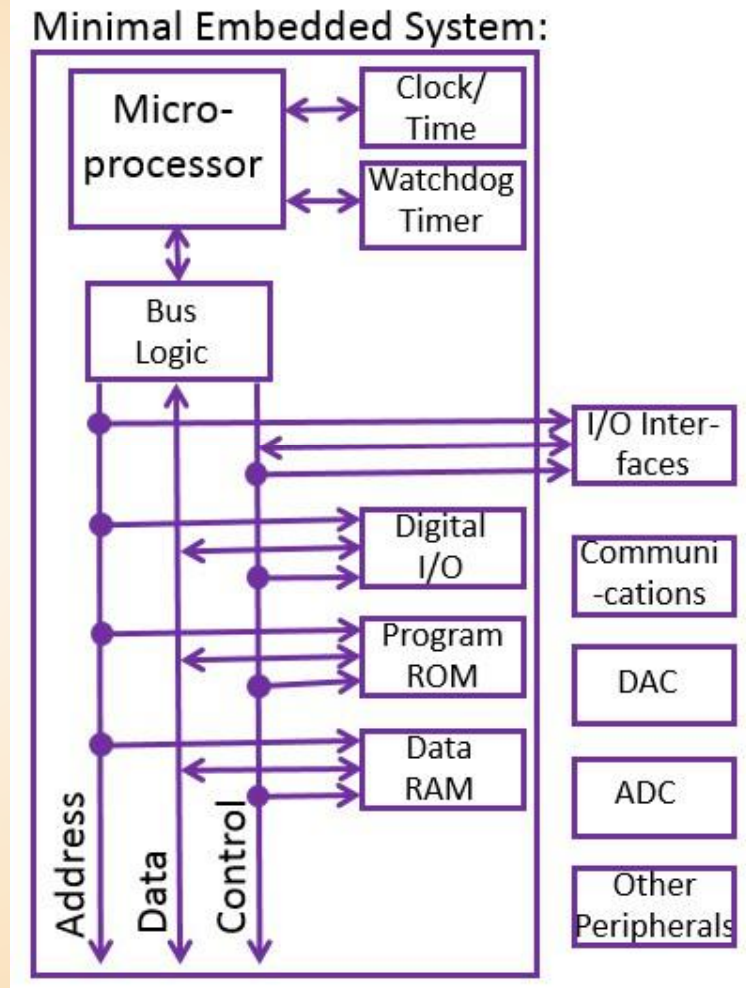


Figure 1.2 Data Movement over an Eight Bit Bus

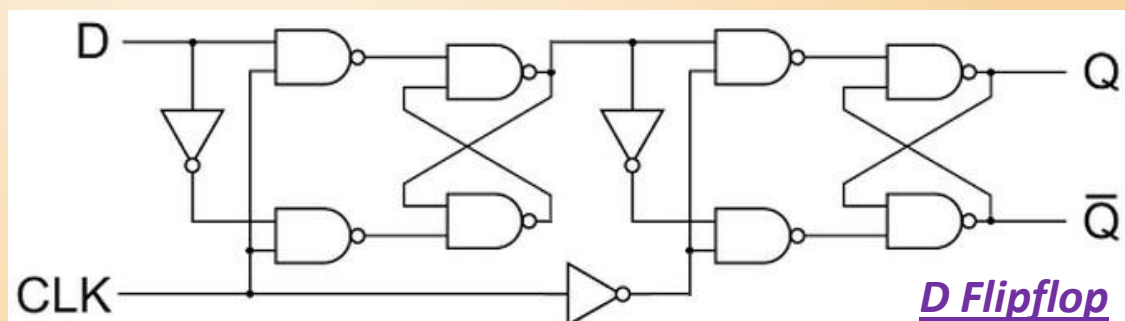
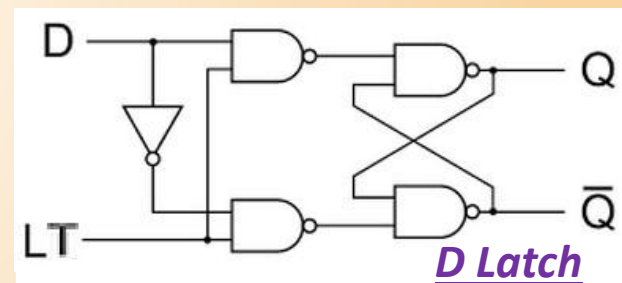
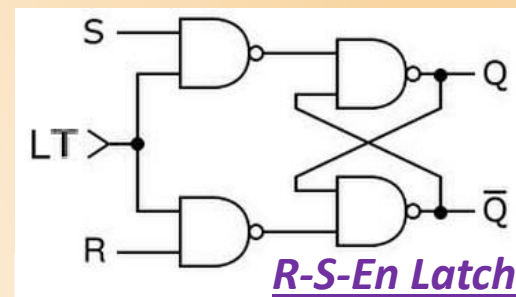
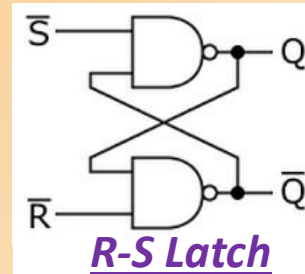
Computer System Vocabulary:

- In the digital world, signals are expressed as collections of binary of 0's and 1 's; the elements of such collections, the 0's and 1 's, are called bits. A bit is simply a variable that takes on either of two values. At the hardware level, a bit may be represented by an electrical signal: a binary 0 as 0 volt and a binary 1 as 5 volts. In an optical communications channel, a bit may also be expressed by the presence or absence of light.
- Data as well as other kinds of signals come into the system from the external world through the input block
- The output block provides the means to send data or other signals back to the outside world
- The datapath and control block, more commonly known as the CPU or central processing unit, coordinates the activities of the system as well as performs computations and data manipulation operations necessary to executing the application



Flip-flop vs register vs latch

- Flop-flops are typically edge-triggered, updates synchronized to a synchronizing signal called a clock. The timing of input capture and output updates is determined by the occurrence of a clock edge.
- Latches outputs can respond immediately to data input changes, though involve an enable signal that is typically level-sensitive.
- "register" typically refers to a multi-bit flip-flop: A 16 bit register is made up of 16 flip-flops in parallel, all triggered by the same clock edge.

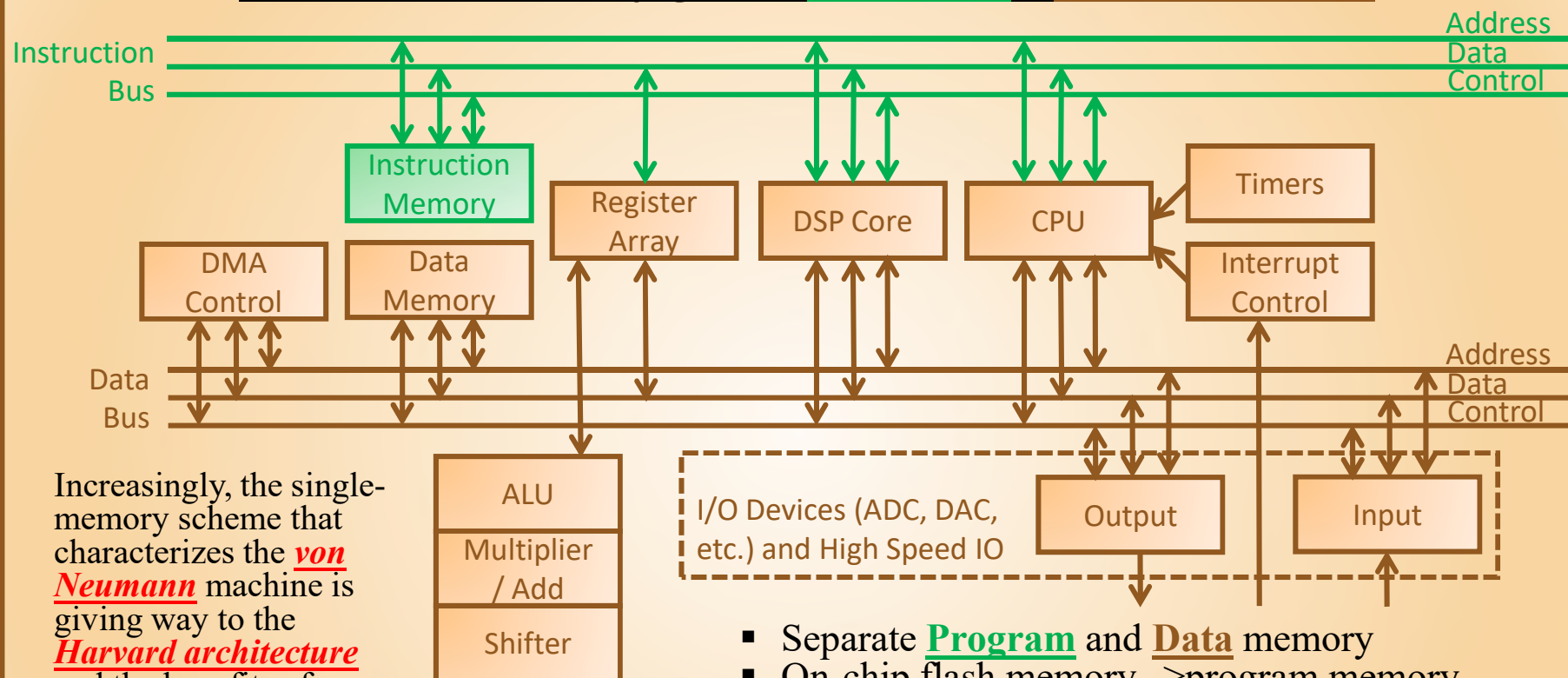


Microprocessor Vocabulary

- Registers are small amounts of high-speed memory that are used to temporarily store frequently used values such as a loop index or the index into a buffer. (what's the diff between internal and external registers?)
- External to the microprocessor, we see two different memory blocks (see Figure 1.7 on the following slide – for the moment, we're just using this as a illustration of how the memory may work. We'll give the same illustration in a later lecture; you will need to know this architecture in more detail at that point). The firmware, or program store, contains the application code, and the data store contains data that is being manipulated, sent to, or brought in from the external world. In the embedded world, we refer to the application code as firmware because it is generally stored in a Read Only Memory (ROM), rather than on a hard drive as one might do for a desktop application. The data memory is usually made up of Random Access Memory (RAM).

Microprocessor Vocabulary

AVR Architecture (fig 1.7): *Harvard* vs *Von-Neumann*



Increasingly, the single-memory scheme that characterizes the **von Neumann** machine is giving way to the **Harvard architecture** and the benefits of simultaneous instruction and data access.

Caution: The two separate pieces of memory do not change the architecture from von Neumann to Harvard unless two separate busses are connecting them to the processor.

- Separate **Program** and **Data** memory
- On-chip flash memory-->program memory
- On-chip data memory (RAM and EEPROM)
- 32 x 8 General purpose registers
- On-chip programmable timers
- Internal and external interrupt sources
- Programmable watchdog timer
- On-chip RC clock oscillator (more on clock later)
- Variety of I/O, Programmable I/O lines

Microprocessor Vocabulary

- *Microprocessor*: an integrated implementation of the central processing unit portion (control and arithmetic and logical unit) of the machine; it is often simply referred to as a CPU or datapath. Microprocessors differ in complexity, power consumption, and cost.
- *Microcomputer*: complete computer systems that uses a microprocessor, though not typically referring to a mainframe
- *Microcontroller*: integrates, memory, IO, communications, and other peripherals and peripheral interfaces
- *DSP*: digital signal processor, microprocessor with special hardware and optimized for signal processing, typically lower cost and power for signal processing applications as compared to general purpose processor.

Numbers

A collection of bits has no inherent meaning. The meaning comes from the interpretation and that is defined by the data-type and its storage specification

- Binary

- Binary Number Representation
 - Signed Binary Number Representations

One's complement (negation: INVERTED BITS,	MSBit is sign bit)
Two's complement (negation: INVERTED BITS + 1 ,	MSBit is sign bit)

Numbers

- Endiannes

- "endianness" describes how to interpret and manipulate data, it is the order of bits or bytes in a word
- May specify order stored in registers, order in memory, order of bits or bytes sent over a communication channel like a serial port
- again, pay attention to word order versus bit order, can be **least** significant **bit** first and **most** significant **byte** first

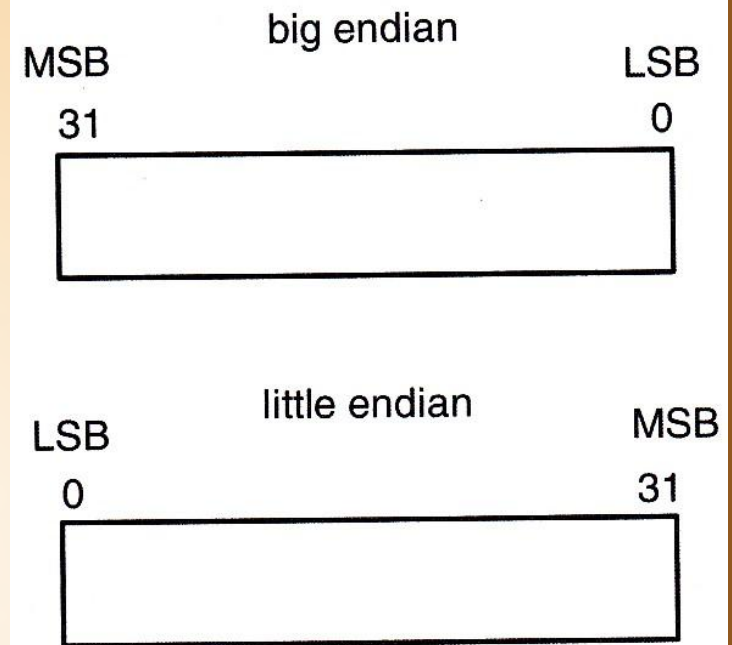


Figure 1.11 Expressing Addresses

	MSB down to	LSBit	
Big Endian	31 ...	0	Most significant bit first, or leftmost; least significant bit last, or rightmost

	LSB up to	MSBit	
Little Endian	0 ...	31	Least significant bit first, or leftmost; most significant bit last, or rightmost

Numbers

- Number of Bits of Resolution
 - Number of bits of resolution relates to the amount of precision and range
 - is the number of levels that can be represented
 - can be given as the # of digits (bits if binary)
 - 7 bits of resolution means 2^7 (or 128) unique levels can be represented
 - Real numbers must be rounded/truncated or otherwise mapped to one of these discrete and countable number of levels. The mapping is called quantization.
 - Error Propagation (possibly more on this later)
 - Addition: error of operands adds
 - Multiplication: error of operands multiply

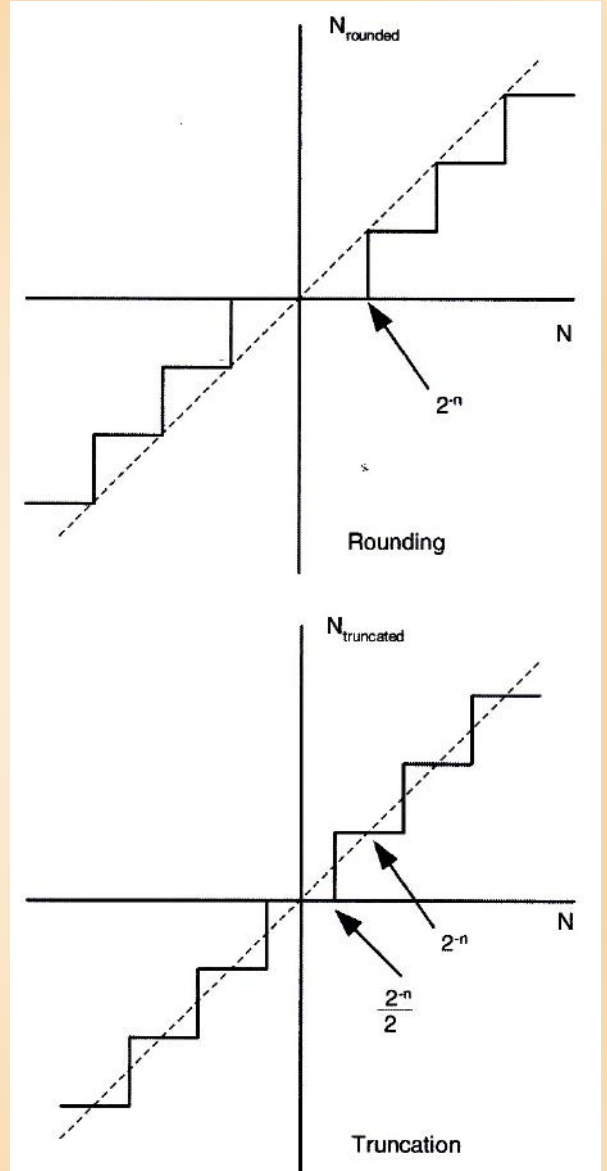


Figure 1.9 Truncation vs. Rounding

Instructions

- *Address* (noun): value used to indicate position in an array
- *Instructions*: used to indicate *actions*
- *Operands*: provide parameters for action
- *Arity* refers to the number of operands an instruction or operation requires
 - e.g. $y+z$; addition here is binary operator
 - Example references to arity: Unary operator, two-operand instruction, two-address instruction
 - Example:
 - $x=y+z$ is a three-operand or three-address instruction
 - Has two sources or source addresses
 - Has one destination or destination address
- Typically instructions are encoded using a concatenation of fields (groups of bits with associated meaning)

Instructions

- Common to have operation or op-code fields concatenated with source and/or destination fields

	Bit 31bit 0	
	operation or opcode operand	
	operation op0 op1	
	operation op2 op1 op1	

- Operands may refer to registers or other memory (or other addresses if using memory-mapped I/O)
- In most systems, if an operand is memory location the contents must be moved to registers to perform the computations
- Opcode design
 - RISC (Reduced Instruction Set Computers) require fewer bits for the op-code (leaves more bits for operands)
 - CISC (Complex Instruction Set Computers) require more bits for the op-code (leaves less bits for operands)

Instructions

- Instruction Set Architecture (ISA) is the interface provided/presented to the programmer/compiler. It is everything that documents the use of the processor but not necessarily the internal details that are of no consequence to the programmer/compiler. Includes specification of op-codes, registers, operand formats, memory access methods, etc...
- Machine Language: collections of 0's and 1's that tell the computer what to do, op-codes and operands
- Assembly Language: programmers create code with instructions chosen from the supported instruction set
- Compiler converts this to machine code --not quite 1-to-1 mapping to machine code but there is a very strong connection from architecture through machine language specification to assembly language and the ISA.

Ex:	add reg1 reg1 add reg1 memaddress1 add reg1 const1
-----	--

We can see "add" requires two operands, in this case we expect reg1 is a source and a destination. Which forms are supported in the assembly language usually relate to what hardware features are available and what machine code operations are supported. (some might support only reg operands, so the last operation would require a load of const1 to a register first)

- Compiled Code: high-level code is converted to assembly through compilation and then to machine code, in contrast to interpreted code

High-Level such a C -> [Compiler] -> Assembly Code -> [Assembler] -> Machine Code

Instruction Set - Instruction Types

- We will discuss the various common types of instructions. Though they vary from processor to processor, the similarities make learning one after another fairly painless.
- Common Instruction Types:

1) Data transfer	transfer and store data
2) Control Flow	Make decisions (based on provided operands or based on "side effect " from another operation (type 1 or 2) or based on a status register byte or bit)
3) Arithmetic and Logic	Operate on data

- The source and destination can be a
 - register,
 - a memory location,
 - or an input or output port

Instruction Set --Instruction Types

- *Addressing mode* determines how to interpret the value provided by the operands. The addressing mode can be specified by the opcode (different opcode for every address mode) or specified by a prepending data field in the operands
 - *Immediate* - data is provided in instruction or operand is the data
 - "add a 4" ("a=a+4") : 4 is the immediate data provided in the assembly and machine code.
 - *Direct* - operand provided by the address in memory of data where the data is stored (or to be stored) instead of the data itself.
 - I write your grade on board in a room. You want the data. I provide you the room number. The room number is like the direct operand.
 - *Indirect* - operand provides address in memory of the address in memory where the data is stored (or to be stored) in memory
 - In the previous example, I instead provide you a web page address where the room number will be posted.
 - With indirect, the location of the information (the grade) itself can change, but you still start in the same place to find it. In the previous example, I can move the grades to another room and change the webpage.
 - *Register direct* - operand indicates which register the data is in or which register the result should be stored in
 - *Register indirect* - operand indicated which register contains the source/destination address of the data in memory
 - *Indexed Mode* - like indirect but address split into base and offset and are provided by two separate operands. This is convenient for working with (large) data arrays.
 - *Program counter relative mode* - program counter relative addressing is typically used for flow controls (jumps/branches). Operand is typically signed is added to the P.C. to generate a new instruction address

Instruction Set - Instruction Types (cont)

- *Data Transfer*

- These instructions require data or location of data. They move or modify memory.

- Typical Types:


- *Load, LD* - move data from memory to register
- *Store, ST* - move data from register to memory
- *STI, LDI* - store immediate value to memory or register
- *MOVE* - move data between registers or between memory locations (really a copy)
- *Exchange, XCH* - exchange data in operands
- *PUSH/POP* - stack manipulation
- *IN/OUT* - transfer data to or from an input/output port

Instruction Set - Instruction Types

• Execution Flow and Flow Control Instructions

▪ Flow concepts:


- Sequential: default describes using a series of instructions with PC auto incrementing by some amount based on the instruction length/size
- Branch: decision-based flow: if, else, switch, case, etc..
 - P.C. modified based on operands or contents of flag/status registers (or bits of). Includes some type of test
 - Common Tests



E,NE	Equality, inequality
Z,NZ	Zero, not zero
GT,GE	>,>=
LT,LE	<,<=
V	overflow
C,NC	Carry, no carry
N	negative

○ Typical branching instructions:

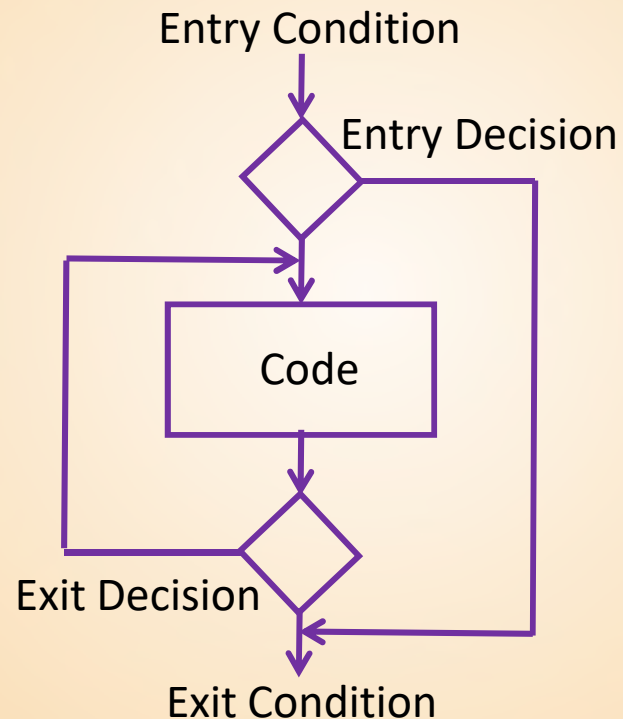
- Unconditional:
 - BR address
- Conditional
 - BE address BNE address
 - also BZ, BNZ, BGT,etc....
- Jumping can be
 - Absolute, PC = something (operand size must support entire program address space)
 - Relative, PC = PC + something (supports jumps with smaller operands)
- If-else example:



```
dec r0
bz _if_label_
ldi r1,0
br _endif_label_
_if_label_:
ldi r1, 10
_endif_label_:
```

Instruction Set - Instruction Types

- *Loop* - construct used to run a block of code repeatedly
 - Structure: Entry Decision/Condition, Exit decision/condition, loop body code



Instruction Set - Instruction Types

- Types of Loops

- Repeat - repeat predetermined number of times
- While - entry decision and exit decision (in code implementation, both may be coded in the same line)
- Do, do While - no entry decision, runs at least once
- For - uses explicit iteration variable that can be used in the code
- Stacks and Procedures / Functions
 - Everyone is expected to know stack, top of stack, push, pop
 - At the low-level, stacks are often implemented in a allocated memory array/block with a stack pointer (SP) that tracks the top of the stack

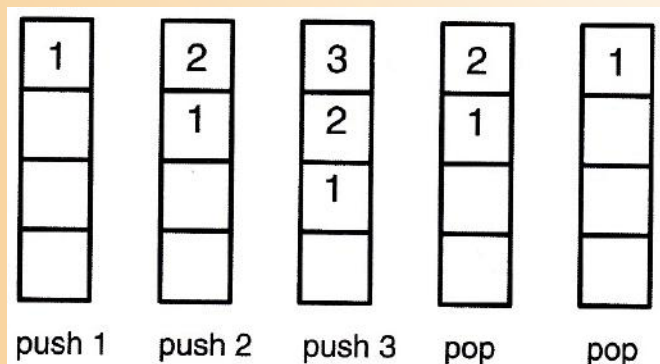


Figure 1.38 Stack Operations

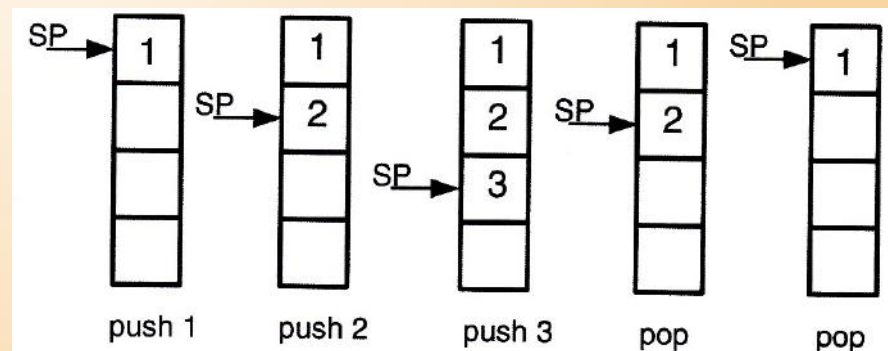


Figure 1.39 Managing the Stack Pointer

Instruction Set - Instruction Types

- *Procedures and Functions*

- Use stack in memory
- Process of a function or procedure call (refer to text for complete descriptions)
 - Push next instruction address into stack or store in memory so we can return to the point just after the call in the code
 - Push parameters
 - Set PC to start of function/procedure code
 - Pop parameters
 - if returning values using stack, must pop instruction address and save in register/memory since return values will bury the return address
 - Do procedure/function work
 - Push return values if function
 - Set PC based on saved instruction address if a or pop off stack
 - Continue execution after original calling point
- Example on Page 30
- Supported through CALL and RET assembly statements

Instruction Set - Instruction Types

- *Arithmetic Operations*

- add, sub, mul, div
- addc - add with carry also adds carry bit, supporting long additions)
- subb - subtract with borrow uses status register/bit to support long subtractions)
- inc, dec
- test -operand(s) tested and results affect status flags but result not saved to registers
- *Comparison Instructions in Intel ASM*
 - **test** arg2, arg1 - Performs a bit-wise AND on the two operands and sets the flags, but does not store a result.
 - **cmp** arg2, arg1 - Performs a subtraction between the two operands and sets the flags, but does not store a result.
- testset - atomic test and set (will learn about atomic operations later)

- *Logic Operations*

- Bitwise operations , e.g. and, or, xor, not or inv
- set and clear bits in registers, e.g. CLR, SET
- set and clear bits in status registers, e.g. CLRC, SETC

Instruction Set - Instruction Types

- Shift Operations: Types are logical, arithmetic (sometimes called a signed shift), rotate (on next slide)

- Logical

- SHR - Fills on left with 0's, discards bits shifted out on right

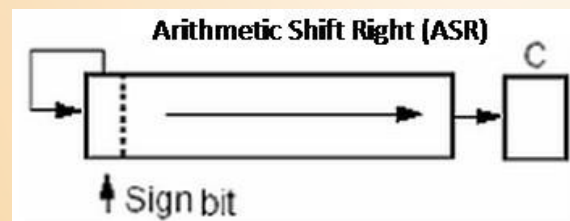


- SHL - Fills on right with 0's, discards bits shifted out on left

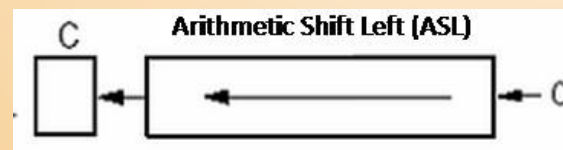


- Arithmetic

- May affect flags like carry, overflow and underflow
- SHRA



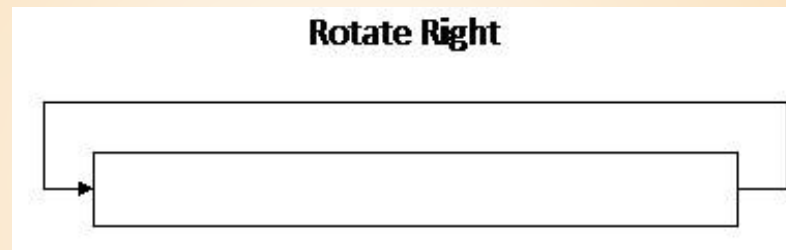
- SHLA



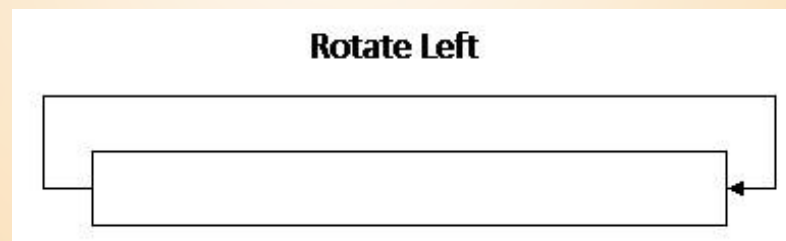
- Fills on left with copies of original sign, preserving the sign of the operation. Discards bits shifted out on right
- Can almost mimick integer divide by power of two except result is rounded towards negative infinity instead of 0, so it is the same for positive numbers but slightly different for negative numbers (-1 shifted right arithmetically by 1 is -1 instead of 0)
- same as SHL but may affect status flags
- can mimick mult by power of two

Instruction Set - Instruction Types

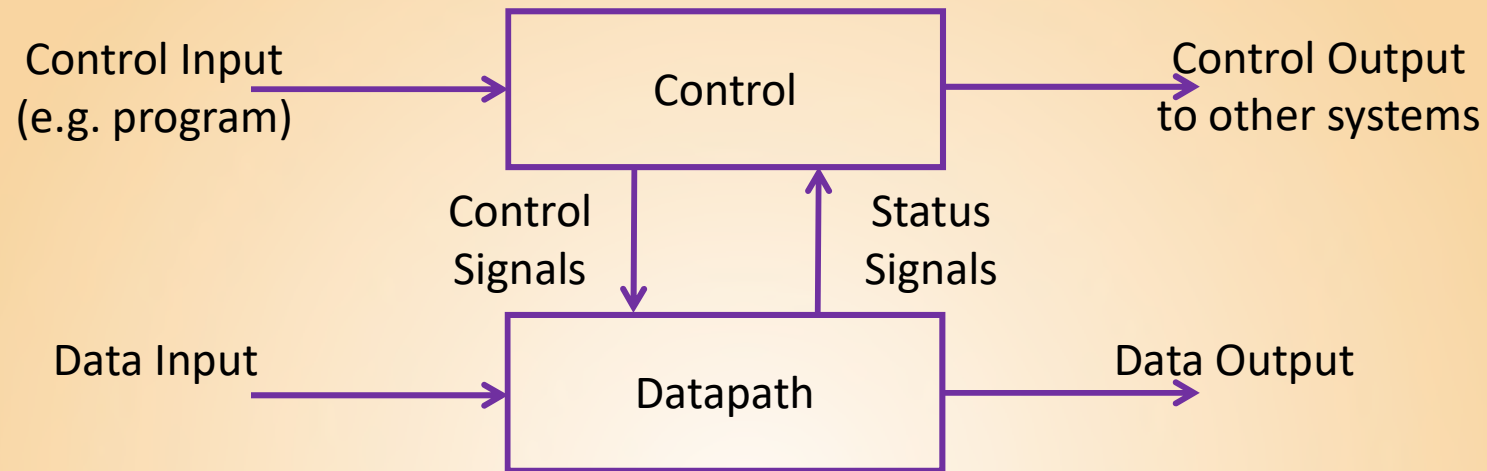
- Shift Operations (continued): Types are logical, arithmetic (sometimes called a signed shift), rotate (*continued from previous slide*)
 - Rotate - No bits discarded, instead they are used to fill other end
 - ROR -



- ROL -

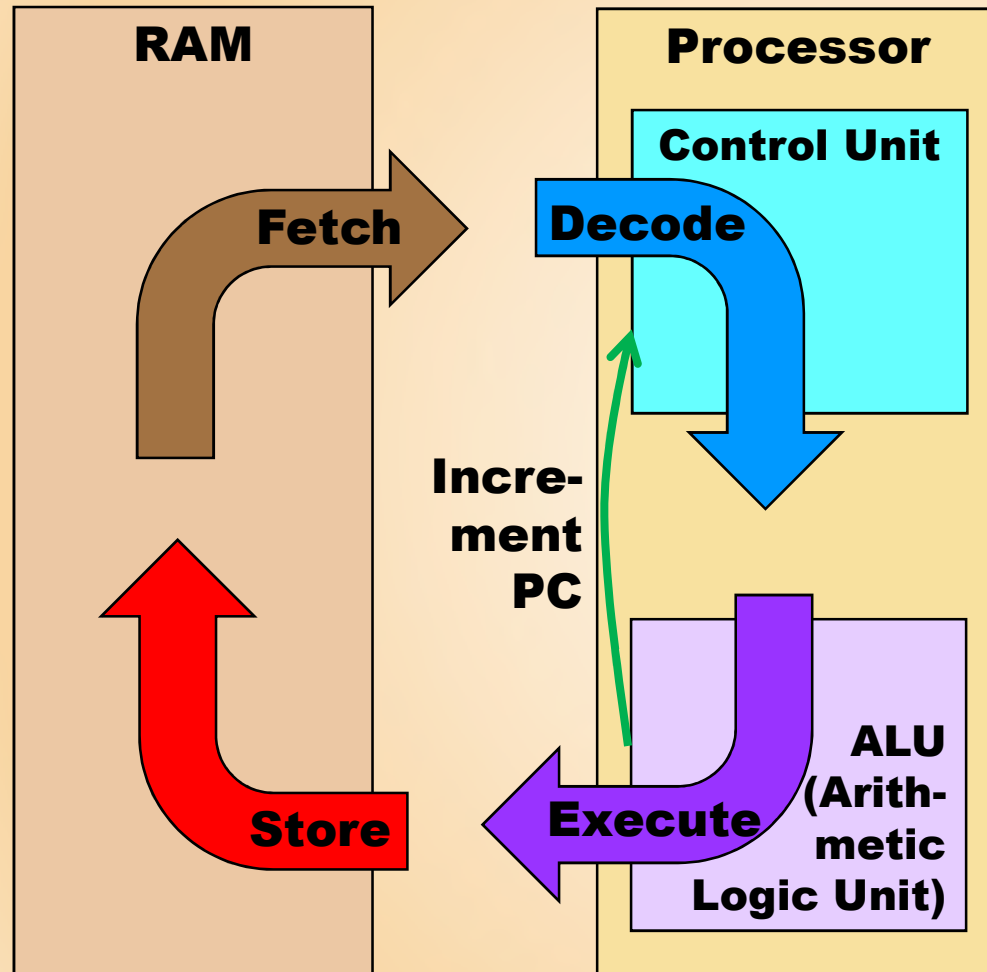


Most digital systems fit into this framework:



- Hardware circuit implementing processor must support fetch, decode, execute, next cycle of code execution and all supported instructions
 - Note, the number of clock cycles required per instruction depends on processor hardware and the operation being performed
- Components and interconnections must support the instruction cycle (see next slide...)
 - Fetch
 - Decode
 - Execute (compute, set)
 - Next (set next instruction address)
- Instruction set determined and limited by hardware design
 - Size and number of decoders, buses, arithmetic circuits, registers, etc. all relate to the supported instruction set

Processor Control – Instruction Cycle:



- **Step 1: Fetch** – the next instruction OS fetched from cache or RAM.
- **Step 2: Decode** – the instructions are decoded into a form the ALU or FPU can understand
- **Step 3: Execute** – the instructions, such as the ALU or FPU performing a computation, are carried out.
- **Step 4: Store (Next)** – The data or results from the instruction execution are stored in registers or RAM. The PC (Program Counter) is incremented to point to the next instruction **Fetch**.

Example from the book: - expresses the architecture of the datapath and memory interface for a simple microprocessor at the register transfer level.

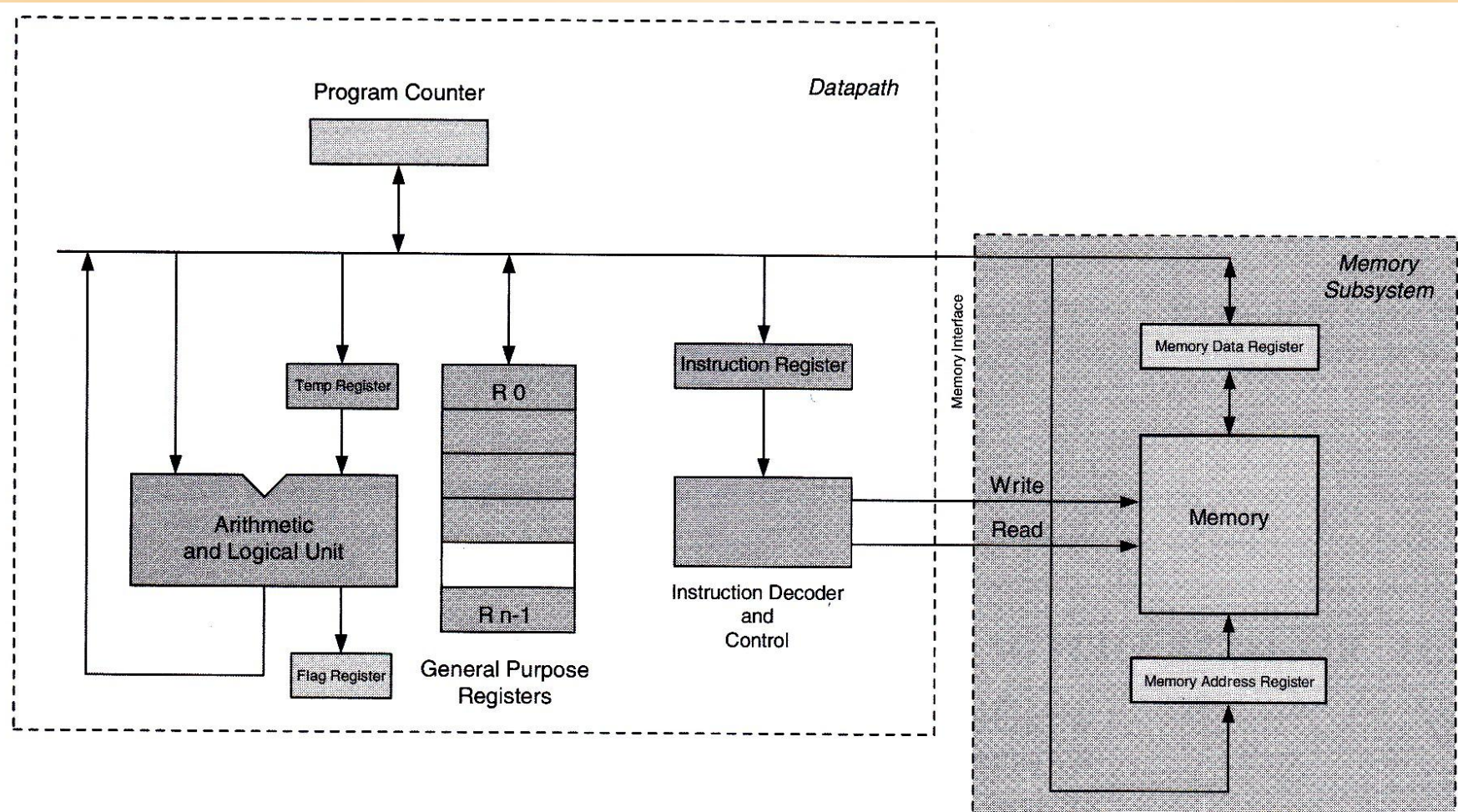


Figure 1.52 RTN Model for a Microprocessor Datapath and Memory Interface

Tabulated Summary of functions:

Table 1.3 Instruction Set Architecture Operations Expressed in Register Transfer Notation

Type	Instruction	ISA Level	Register Transfer Level
Data Transfer	Move register	MOVE R1, R2	$R1 \leftarrow R2$
	Move from memory	MOVE R1, memadx	$R1 \leftarrow (\text{memadx})$
	Move to memory	MOVE memadx, R1	$(\text{memadx}) \leftarrow R1$
	Move immediate	MOVE R1, #DEAD	$R1 \leftarrow \#DEAD$
Control Flow	Unconditional branch	BR \$1	$PC \leftarrow \$1$
	Conditional branch	BNE \$1	$\text{cond} (PC \leftarrow \$1)$ $\text{if}(\text{cond}) PC = \1
Logic	Complement accumulator	CMA	$A \leftarrow \neg A$
	AND register	AND R1, R2	$R1 \leftarrow R1 \wedge R2$
	OR register	OR R1, R2	$R1 \leftarrow R1 \vee R2$
	Shift register	SHL R1, #3	$R1 \langle 31..0 \rangle \leftarrow R1 \langle 31-n..0 \rangle \#(n@0)$ Contents of R1 get replaced by bits in range of 31-n..0, where n is number of bits to shift and n 0s get extended on right
Arithmetic	ADD register with carry	ADDC R1, R2	$R1 \leftarrow R1 + R2 + C$
	Clear carry	CLRC	$C \leftarrow 0$
Program Control	Don't execute an instruction	NOP	$PC \leftarrow (PC + 1)$
	Stop executing instructions	HALT	$PC \leftarrow PC$