Introduction

Embedded systems

Continue pervasive expansion into

Vast variety of electronic systems and products

Have difficulty identifying any products

Not incorporating embedded processor

In one form or another

Different kinds of computing

We identify three basic kinds of component

We embed into our systems

Microprocessors microcomputers and microcontrollers

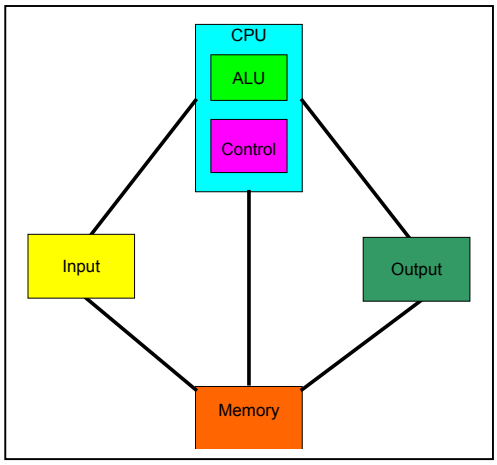
What's the difference

First

Most units CISC architecture

Based upon fundamental von Neumann architecture

Also known as Princeton architecture



- Microprocessor

Integrated implementation of central processing unit – CPU

Given in above diagram

Thus often simply referred to as CPU

Microprocessors will differ in

Complexity

Power

Cost
One may also find differences in
Number of registers
Overall control
Bus structure
Increasingly Aiken or Harvard architecture
Becoming mixed in
In addition RISC features being utilized as well
Microprocessors range from devices with
Few thousand transistors
Cost of a dollar or less
Units with 5-10 million transistors
Cost several thousand dollars

To implement complete computer
Must still include
Input / output subsystems
Memory
Will examine each of these in detail
In context of embedded system

Such components connected via system bus

- Microcomputer

Complete computer
Implemented using microprocessor
Typically constructed utilizing
Numerous integrated circuits
Once again complexity varies
Simple microcomputers
Can be implemented on single chip
These will have limited
Onboard memory
Simple I/O system

- Microcontroller

  Includes

    Microprocessor

    I/O subsystems

      Typically these include such things as

        Timers

        Serial communications channels

        Analog to digital conversion

        Digital to analog conversion

        DMA

  Memory subsystem

    May or may not be included

Embedded Systems an Overview

- An embedded system is a microcomputer system

  Comprising hardware and software

    Designed and optimized

      To solve specific problem very efficiently

- Typically continually

  Interacts with environment

    Monitor and control some process

Term embedded system refers to fact

  Microcomputer system

    Enclosed or embedded

      In larger system

Typical person

  May interact with 10-20 embedded systems

    Around home each day

Single contemporary automobile

  May contain as many as 100 embedded microprocessors and microcontrollers

    Engine ignition

    Transmission shifting

    Power steering

    Antilock braking

Security

Entertainment

Time

Many embedded systems

Considered to be real time systems

Real time system

Must respond within constrained time interval

To external or internal events

Response is execution of task associated with event

Soft real time system

If time constraint not met

Performance degraded

Hard real time system

If time constraint not met

System considered to have failed

Failure may be considered to be catastrophic

When operating system used in embedded microcomputer

Typically is real time operating system

Real time operating (RTOS)

Designed and optimized

To handle strict time constraints

Associated with events in real time context

Architecture

As noted earlier

Components of microcomputer

Connected via system bus

System bus

Provides path for electrical signals

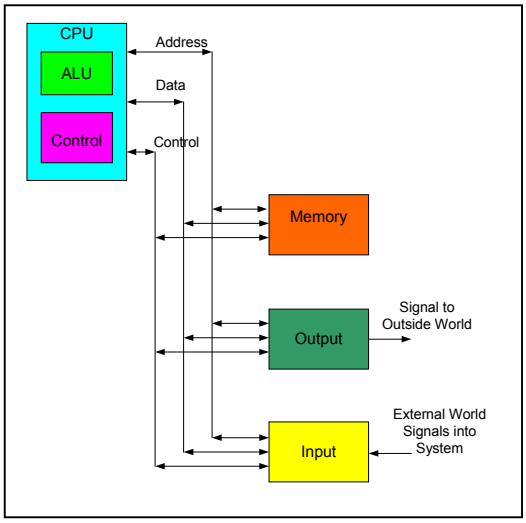
To flow amongst components

Subdivided into three busses

Segregated by function

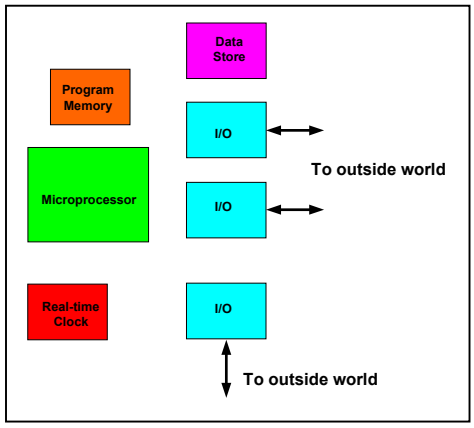
System may now look like following

At high level



Implementation of *microprocessor* embedded system

Takes the following form



Observe

Individual pieces segregated

As the engineer

We design and integrate each of the pieces

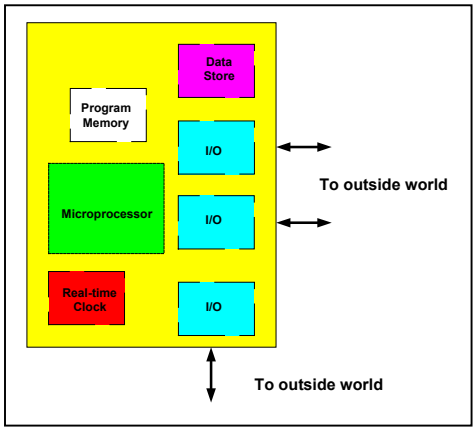
Must have an understanding of and consider

HW point of view

SW point of view

Implementation of *microcontroller* embedded system

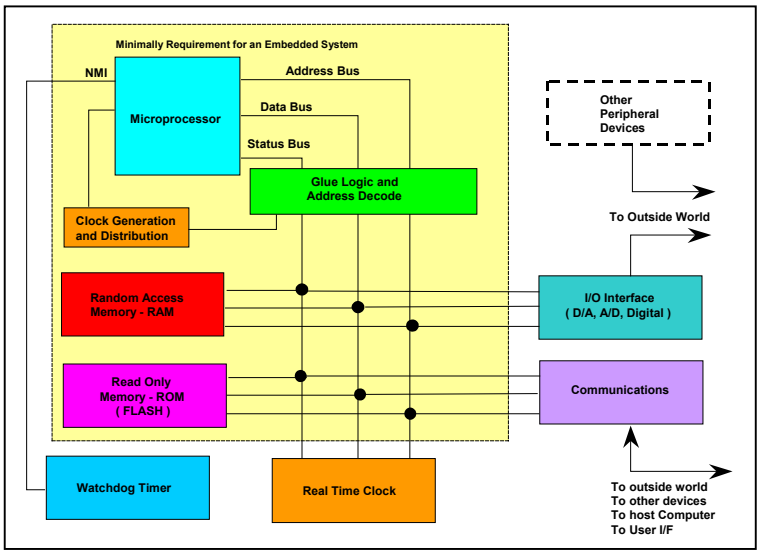
Takes the following form



Observe

Individual pieces combined into whole

We now combine all pieces into complete system



Microprocessor controls whole system

Executing set of instructions

Stored in memory subsystem

Generally stored in ROM type memory

Thus called *firmware*

At power on

Microprocessor addresses

Predefined location

Fetches, decodes and executes instruction

When instruction execution complete

Fetch next one

Process repeats forever

Unlike typical application program

At top-level of embedded program

```
while(1)
{
   embedded program
}
```

Does not execute return

No place to return to

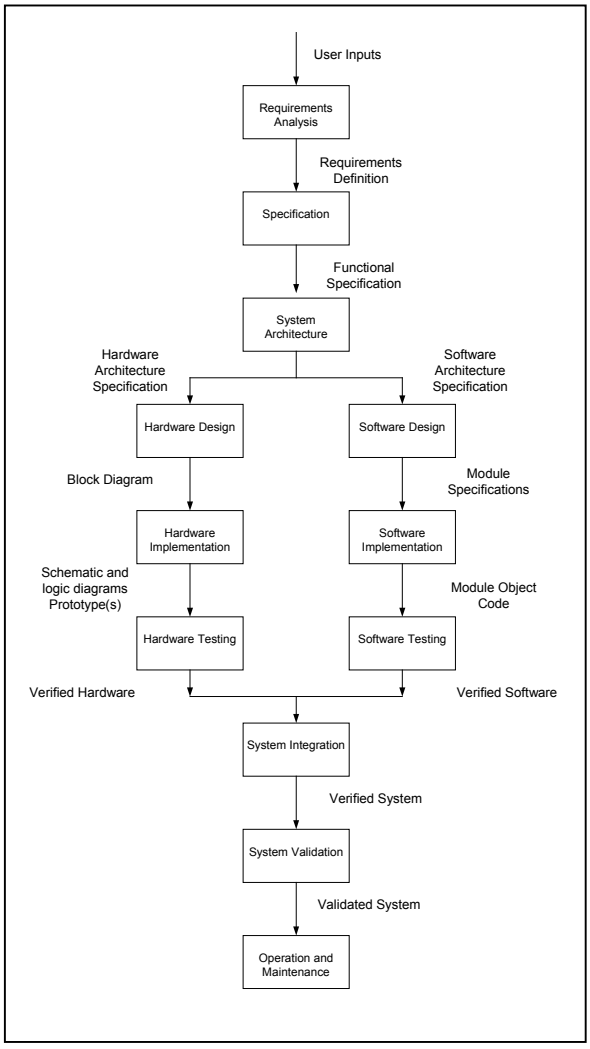
Specific set of instructions for microprocessor
- Called *instruction set*
  - Includes instructions that
    - Bring data in from outside world
    - Output signals to external world
    - Provide means to exchange data
      - With memory subsystem
  - Instructions or control flow
    - Sequential
    - Branch
    - Loop
    - Function call

Embedded Systems Development

As we've seen
- Most embedded systems
  - Share a common structure

Also share a common development cycle
- Through such a cycle
  - Develop the embedded design

Adjacent figure
- Gives high level flow
  - Identifies major elements of such cycle



In subsequent lectures
- Will examine each phase in detail

Specifically
- Hardware design
  - Involves design of
    - Components
    - Printed circuit boards
    - System
- Software design
  - May entail design of
    - High level language code
    - Assembly

Work in assembly

Requires detailed knowledge of

Microprocessor architecture

Its register structure

Mixture of assembler and high level

Embedded Systems – A Register View

Registers one of fundamental elements of microprocessor system

We define 3 basic operations on data in system

All involve registers

Operations

- Store data
- Transfer data
- Operate on data

Using register view of system

Simplifies and aides understanding

Referred to and RTL model

Commonly used as one of our high level models

Basic Register Operations

We express basic register operations

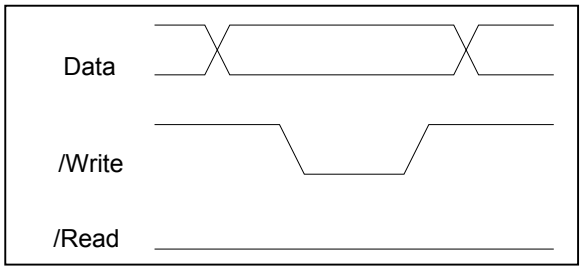
According to following timing diagram

Reflected are

- Read
- Write

All other operations built on these

On *write* operation



Data changed on inputs to register

Following delay

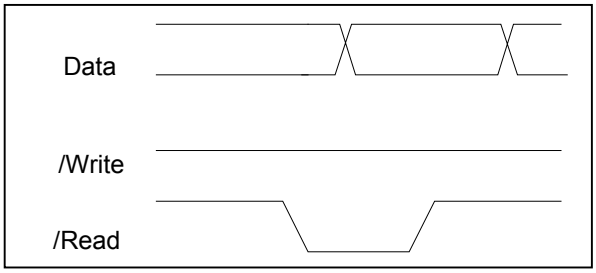
To allow data to settle on bus

Write signal asserted

In drawing signal asserted low

This is typical

Read follows similarly



The read signal is asserted

In drawing

Asserted low

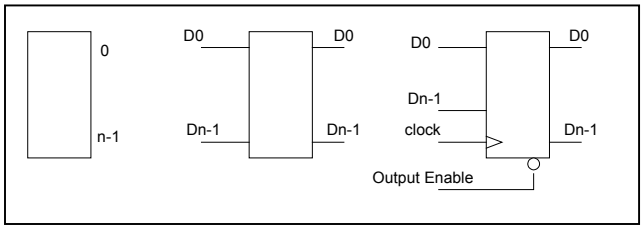
Following some delay

Data appears on output of register

This will be copy of contents of register

Based upon the level of detail we need

We take several views of a register



Simplest view shows simple box

With bits numbered

More complex shows

Inputs and outputs

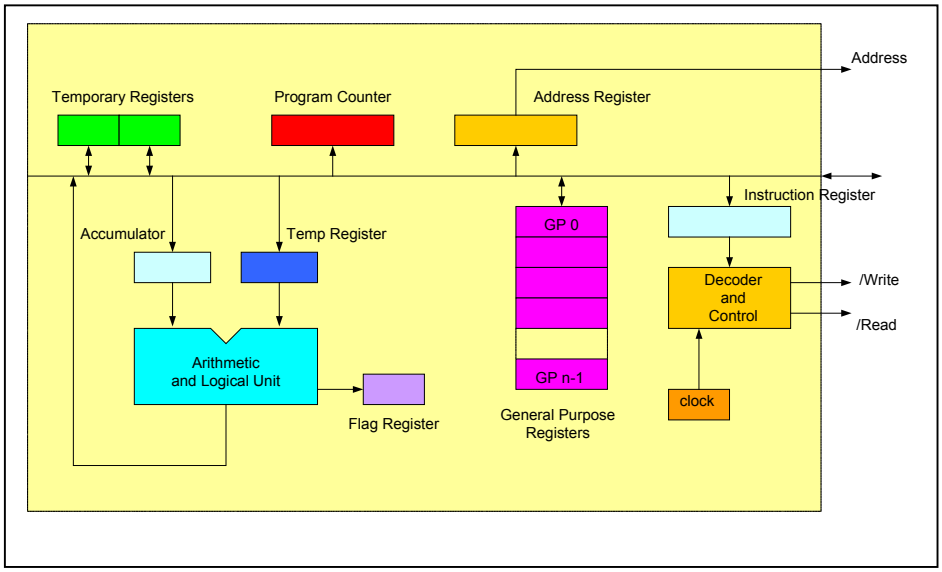
Some control signals

Register View of Microprocessor – The RTL Architecture

Can now express architecture of simple microprocessor

From register point of view

See this in following figure



System Operation

We can now view system operation

From register's point of view

Microprocessor's instruction set

Specifies how we execute the basic operations

On the registers

Once again these are

- Store data
- Transfer data
- Operate on data

Corresponding to such operations

Classify instructions into following groups

- Data Transfer
- Logic
- Execution Flow
- Arithmetic
- Processor Control

We express instructions

Assembly level

Register transfer level

Following table gives representative examples

Each kind of instruction

Type	*Instruction*	*Assembler*	*Register Transfer*
Data Transfer	move register	MOV R1,R2	R1 ← R2
	move from memory	MOV R1,memadx	R1 ← (memadx)
	move to memory	MOV memadx, R1	(memadx) ← R1
	move immediate	MVI R1,#DEAD	R1 ← #DEAD
Logic	complement accumulator	CMA	A ← !A
	AND register	AND R1	A ← A ∧ R1
	OR register	OR R1	A ← A ∨ R1
Execution Flow	unconditional jump	JMP $1	PC ← $1
	conditional jump	J<cond>	if<cond> == 1 PC ← $1
Arithmetic	ADD register with carry	ADD R1	A ← A + R1 + C
	Clear carry	CLC	C ← 0
Program Control	Don't execute an instruction	NOP	
	Stop executing instructions	HALT	

Embedded Systems – A Software View

Computer hardware is a tool for getting job done

As early computer developers showed

By re-wiring computer appropriately

Could configure to solve desired problem

Originally all integer math

Anyone knows how to scale numbers

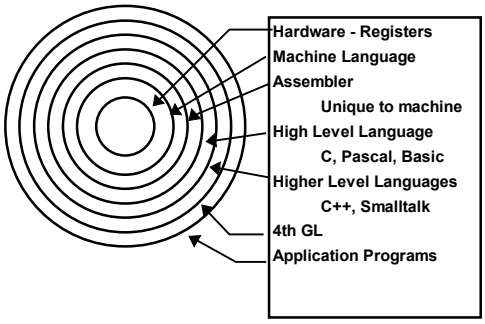
Quest for making job easier

Increasingly sophisticated and powerful languages developed

Machine language

Assembler

Higher level



Can use familiar onion model to view relationship

Begin at center with hardware

Move to outside to point and click languages

Combining Hardware and Software

As we've noted

Embedded computer and hence system

Is a tool we've used to make solving problems easier

Requires understanding of both hardware and software

Let's follow the process from problem statement

Through a computer to see how it works

Problem

Let's begin with simple problem

Control brakes to prevent locking the wheels

Problem is stated in natural language

Our embedded system

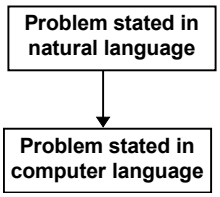
Cannot accept problem in such a form

1. Must translate into more compatible form

   We will see that all our activities really involve

   Translating problem from one form to another

   Until we reach representation we can solve



In current case

We translate into some new language

C

Basic

This translation is done by hand

Note at this point

We can have a variety of translations

They are not unique

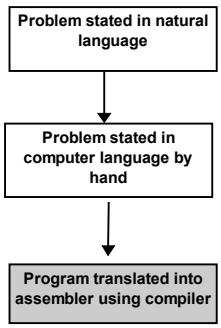
This translation is what we call software design

Or in embedded case this turns into firmware

Result is what we call a program

Let's see where we are now

2. More translation



This still not enough

We require addition levels of translation

At this stage

We can begin to use some additional tool to help in process

First such tool is called compiler

Compiler

Compiler is a tool for translating programs

Into variety of forms

Comprises several different tools including

Assembler

Linker

One such form

Assembly language

Observe prior to this stage

Program did not depend upon machine

Now program in form that will execute only on particular machine

In embedded case

We work with a cross compiler

Compiles on one machine for a second or target machine

Assembler

Assembler is next tool we use for translating

Converts assembly language into machine language

Program expressed as collection of 0's and 1's machine understands

Linker and Loader

Although program now in machine language

Not ready to be executed

Problem

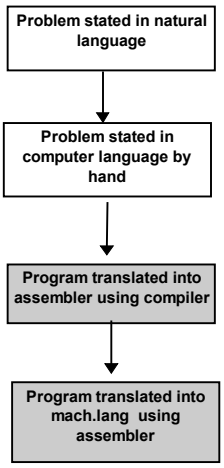
All variables and data structures we use

Must reside in computer memory

Each needs an address in memory

Question

Which address should we use



Unfortunately

Cannot always use the same address

What if someone else wants to use same address

Typical for desktop PC

In embedded application

Want to put at specific address

To solve problem assembler generates

Relocatable code

Code that can be placed anywhere in memory

Second question arises at this time

We'd like to be able to use existing code

Our own

Other people's

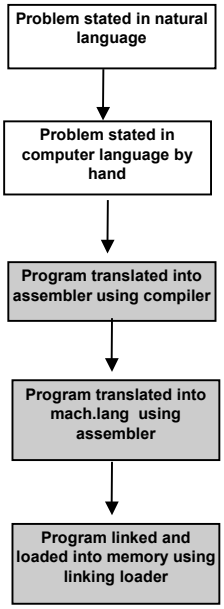
How do we get this into our program without typing in each time

Tool called *linker loader* can help with both problems

Does two jobs

1. Links collection of program modules together
2. Resolves address problems

Adding that step to flow we now have



3. Into Memory

Not quite there yet

Several more stages to go

At this point linker and loader have gotten program into memory

PC systems this means onto the hard drive

Embedded system this means a ROM

We will see shortly

Memory actually comprised of hierarchy of elements

On desktop computer these can include

Hard drive

RAM

CACHE

Registers

Instruction register

In embedded application we have

ROM

RAM

CACHE

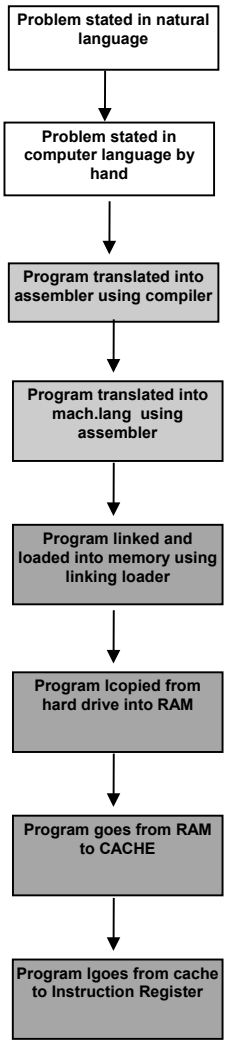
Registers

Instruction register

Adding these we come up with

We have now seen how to take problem
- Expressed in natural language
- Turn into something that can be solved by computer

Throughout remainder of quarter
- Will examine all steps in greater detail



Embedded Systems – A Firmware View

Generally we write embedded system program
- High level language
  - C, C++
- Assembly language
  - For the machine on which we're working

Sometimes we write combination
- Done when portions of programs
  - Optimized for
    - Speed
    - Size

When working with assembly language
- We work directly with microprocessor's
  - Various registers
- Consequently
  - Must have knowledge of machine's architecture

Assembly language instructions
- Translated into *machine code*
  - By assembler
- Machine code reflects binary encoding
  - Machine's instructions

Anatomy of a Machine Instruction

At highest level machine instruction

Machine instruction contains following information

Operation the instruction is to perform

Referred to as *operation code*

Shortened to *op-code*

Operand information

Operands are pieces of data the

Microprocessor is operating on

Typical machines support instructions

Involve

One, two, or three operands

Known as

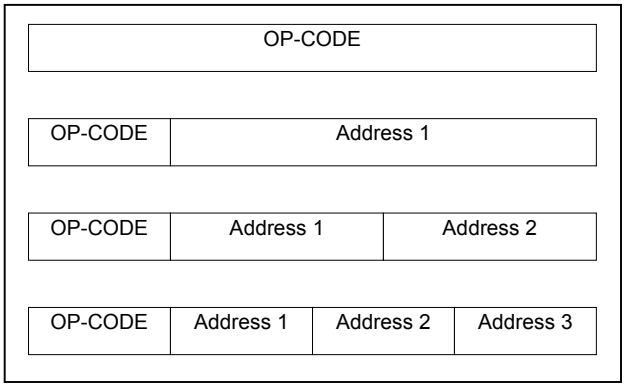
One, two, or three operand instructions

Each such operand has an address

Consequently also known as

One, two, or three address instructions

We depict these graphically as shown



Observe that we have one instruction

With no apparent address

Such instructions support actions such as

NOP – no operation

HALT – stop executing

Referred to as zero address instructions

Remaining instructions interpret addresses as

Source of operand

Destination of result

Code Fragment	*Operand 1 (Source 1)*	*Operand 2 (Source 2)*	*Result (Destination)*	*Type*

int a = 10;
int b = 20;
int c;
c = a + b;
	a	b	c	3 address

int a = 10;
int b = 20;
b = a + b;
	a	b	b	2 address

a = ~a	a			1 address

To support full address of operand

Address fields of instruction must necessarily

Be very large

For 16 bit machine

Support for 3 address instruction would necessitate

48 bits for address

op-code bits

While certainly feasible

Alternate schemes possible

Driven by need for speed

Flexibility

Interpretation of address at

Compile time vs. runtime

Alternate interpretation for bits in address fields

Provides for speed and flexibility

Designate subset of bits as address mode

Let remainder specify address

Addressing Modes

Commonly implemented addressing modes include

- Immediate

  Operand is part of instruction

```
LOADI R1, #FACEh;          // put hex constant FACE into register R1
```

- Register direct

  Named registers contain operand

```
ADD R1, R2;          // add contents of R1 to contents of R2 and put result
                     // into R!
```

- Register indirect

  Named register contains address of operand

```
ADD R1, *R2;         // add contents of R1 to contents of what R2 is
                     // pointing at and put result into R!
```

- Indexed

  Address computed as sum of

  Base address

  Contents of indexing register

```
ADD R1, 1800(R2); // add contents of R1 to contents of memory location
                  //  1800 + contents of R2 and put result into R!
```

- PC relative

  Signed offset added to PC

  Becomes address of next instruction

```
BR -10;              // add  –10 to contents of PC go to that location
```

Different vendors implement

Addressing modes in different ways

May have variations on fundamental scheme

Flow of Control

We can now use various addressing schemes

To alter the flow of control through program

Let's look at alternatives at high level first

High Level View

We have 4 basic ways we can proceed through program

Sequential

Each instruction executed in sequence

Branch

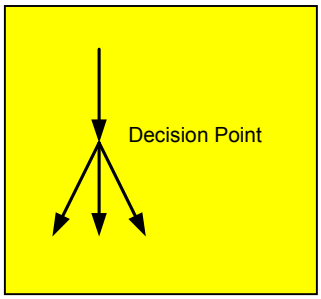
Select one of several branches based upon condition

Graphically

Type of construct seen in

if else

switch or *case*



Loop

Repeatedly execute set of instructions

Forever

Until some condition met

Can make decision

Before

Code may not be executed

After loop

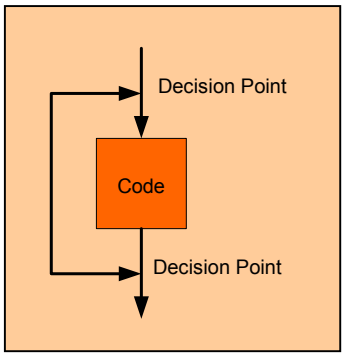
Code executed at least once

Type of construct seen in

do or *repeat*

while

for

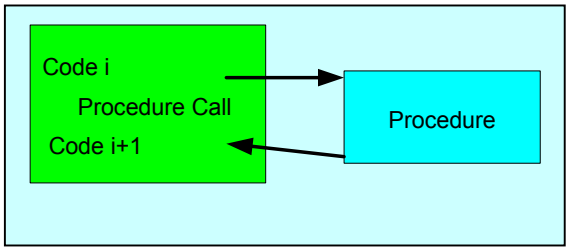


Procedure or Function Call

Leave current context

Execute set of instructions

Return to context



Type of construct seen for

Procedure or subroutine call

Interrupt handler

Co-routine

Implementation

Let's now look at the assembly language level

Sequential

```
a = 10;
b = 20;
c = a + b;

LD R1, 10          // puts 10 into R1
LD R2, 20          // puts 20 into R2
add R3, R1, R2     // computes R1 + R2 and puts result into R3
```

Branch

if - else construct

```
if (a == b)
    c = d + e;
else
    c = d - e;

Assume a .. e in registers R1..R5

    CMP R2, R1       //  test if contents of R1 and R2 equal
    JE $1            //  if equal branch to $1

                     //  $1 is a label created by compiler
    SUB R3, R4, R5   //  compute d - e and put results in c
    BR $2            //  $2 is label created by compiler
$1: ADD R3, R4, R5   //  compute d - e and put results in c
$2: ...
```

Loop

```
while (a < 10)
{
    i = i + 2;
    a++;
}
```

Assume a in R2 and i in R3

```
$1: CMP R2, 10      //  test if R2 < 10
    JGE $2          //  if R2 greater than or equal 10 branch to $2
    ADD R3, 2       //  compute i + 2 put result in i
    ADD R2, 1       //  auto increment a note an int is 4 bytes
    j $1            //  continue looping
$2: ....
```

Procedure Call

Most complex of flow of control constructs

Not more difficult

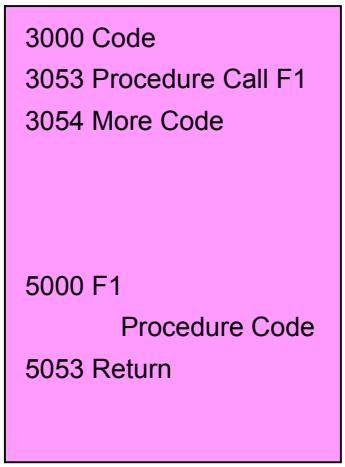
More involved

Will include

Procedures

Subroutines

Co-routines



Process

We'll consider from high level

Program loaded at address 3000

Code executed until address 3053

Procedure encountered

1. Save return address

   Several important things to note

   Address saved is 3057

   Stack gets

Return address
Parameters

2. Address of procedure 5000 put into PC
3. Instruction at 5000 begins executing
4. Execution continues until 5053
5. Return encountered
    - Action similar to call
    - Stack gets
        - Return values
    - Stack looses
        - Return address
6. Return Address put into PC
7. Execution continues at 3057

- Had procedure call been encountered in procedure F1
    - Identical process repeated
        - Can be repeated multiple times
    - Must be aware that stack can overflow
    - If too much pushed on
        - Begin to loose information
            - Particularly return address

Stack

- Stack data structure
    - Occupies an area in memory
    - Has finite size and several operations

    - *Push*
        - Puts something onto top of stack
            - Top of stack is special value
                - Saved for ongoing operations

        - Push operation
            - Writes something to memory
            - Increments address of top of stack
                - Ready for next push

For ease of implementation

Stack typically implemented from

Lower memory addresses to higher memory addresses

Pop

Takes something off the top of stack

Pop operation

Removes something from top of stack by

Decrementing top of stack

Returning previous top of stack

Peek

Looks at something on the top of stack

Peek operation

Returns something from top of stack by

Does not decrement top of stack

Stack Frame

Item being pushed onto or popped from stack

Data structure called stack frame

Special area set aside in memory for such purpose

Will not go into details at moment of

Implementation

Handling

Sufficient to say

Contains

- Return address
- From calling context

  Register values

  Local variables

  Passed parameters
- From return context

- Values to be returned

- Management
  - Created when procedure called
  - Pointer
    - To stack frame returned
    - Pointer placed on stack

Expanding Op-Codes

- In discussion above
  - We've assumed op-code field fixed
- No reason for this to be the case
  - Consider high level strategy
    - Similar to encoding data for transmission
      - Make most frequently sent the simplest
    - At outset of design
      - Designers examine necessary capabilities for system
      - Entails identifying
        - What kinds of instructions to support
        - How many of each kind
          - Recall our earlier discussion
        - How many of each operand type
          - May need only few 1 address instructions
          - Many 2 address versions
          - Somewhat fewer 3 address implementations
    - Forcing all op-code fields to be same width
      - Wasteful

- Consider logically subdividing op-code field
  - Into subfields

- Devote portion to encoding intended operation
- Let remainder distinguish amongst types

- Now consider how we might interpret such a field
  - One possibility

Assume we need a large number of 2 address instructions
Let MSB distinguish between
Two address (operand) instruction
Not two address instruction
Second MSB or two MSBs
Distinguish amongst remaining types

Now we can specify
If two address instruction
Most significant m bits express op-code
If not two address instruction
Most significant k bits express op-code
As designers
We have perfect knowledge
What bits represent
How to interpret them

One Bus Architecture

Let's now take one step further and see how all this works
Inside the microprocessor

From name we see design has single bus

Made up of

- Data
- Address
- Control

Only one device can be driving the bus at any time

Multiple devices can be simultaneously listening

At top

Instruction Register - IR

Holds instructions fetched from memory

Program Counter

Identifies the next instruction to be fetched from memory

Registers R0 - R31

Hold information

Y, Z

Temporary registers

ALU

Arithmetic and Logical Unit

Does computation

Memory

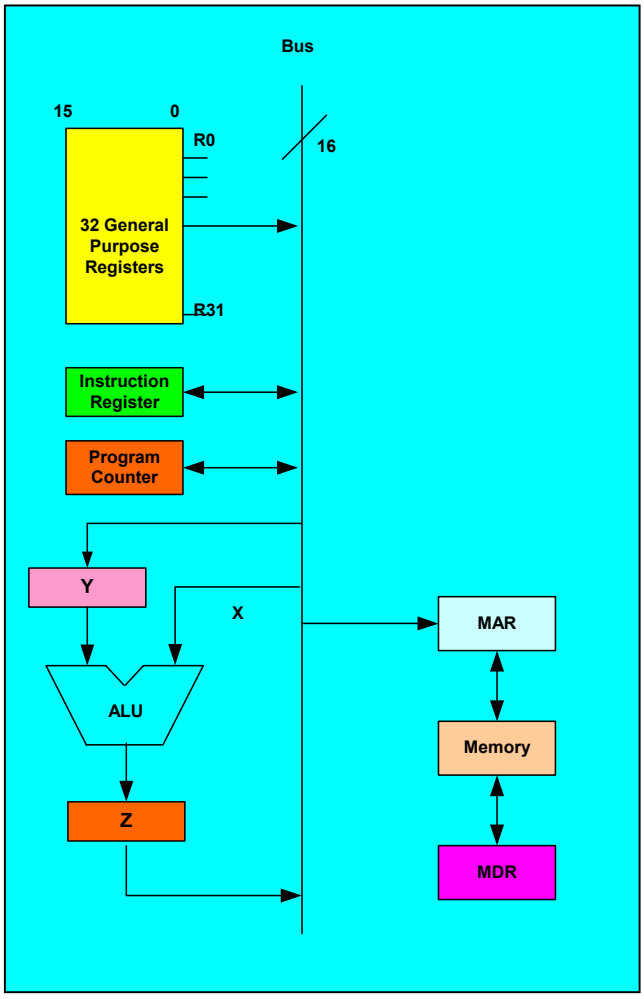
Stores instructions and data

Memory Address Register - MAR

Holds address or instruction or data in memory

Memory Data Register - MDR

Holds data to be read from or written to memory

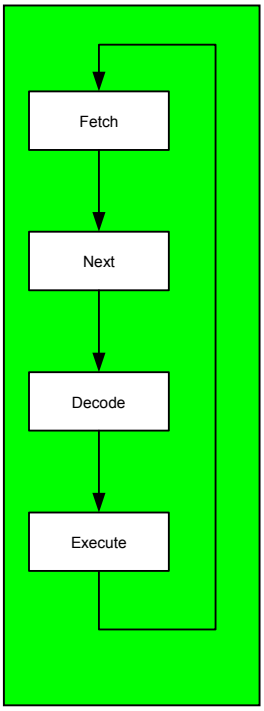


Instruction Execution Cycle

Instruction execution cycle comprised of 4 basic steps

Fetch	Fetch instruction
Next instruction	Compute address of next
Decode	Decode current instruction
Execute	Execute current instruction

Can describe according to following state diagram



Operation

Fetch

Place address of instruction from PC onto Bus
Store contents of BUS into MAR
Issue a READ command
Place contents of memory location into MDR
Place MDR onto BUS
Store contents of Bus into IR

Next

Place contents of PC onto Bus
Place 1 into Y register
Add contents of Y and Bus in ALU
Place output of ALU into Z register
Place contents of Z register onto Bus
Store contents of Bus into PC

Decode

Decode OP Code field

Execute

Do the sequence of steps to perform the instruction

Let's look at the sequence of operations necessary to execute a simple instruction

In assembler

ST *R1, R2

This is a move indirect instruction

Use the contents of register R1 as an address in memory
Read what's at that location in memory
Place the value in R2

Recall the instruction cycle

Fetch, Next, Decode, Execute

Let's assume we've done the fetch part and the instruction is in the IR

We now first walk through the sequence of operations

Note the control signals that are active at each step

We read / write from/to a register by

Selecting it

Issuing the Read / Write control signal

We assume all such actions are synchronized to the system clock

Next

Place 1 on input to Y register
Write to Y register
Read to PC register
Command ADD to ALU to Add contents of Y and Bus in ALU
Place output of ALU on input to Z register
Write to Z register
Read from Z register
Write to PC

Decode

Decode the opcode ST
Routed to collection of control logic called microprogram
Knows how to execute each assembler instruction

Execute

Read from R1 onto the Bus	// Instruction contains adx of R1
	// and info for indirect addressing
Write to MAR	// Specified in microcode for ST
READ to memory	
Wait	
Write to MDR	
Read from MDR	
Write to R2	// Instruction contains adx of R2
	// and info for direct addressing

Observe that we've indicated all actions happen in strict sequential order

In fact number of operations can be combined to happen at same time

Provided actions don't require same resource

- Cannot

  Read from two different sources at same time

  Both require bus

- Can

  Write to two different destinations

  Read from one and write to another

Summary

We've looked at

High level view of computing elements

Distinguished

- Microprocessors
- Microcomputers
- Microcontrollers

Introduced the concept of embedded systems

Taken an overview of such systems

Seen how time plays role in

Behaviour of such systems

Notions of

Soft real time

Hard real time

Explored system architecture

Distinguished

Hardware

Software

Firmware

Seen the embedded systems development cycle

Examined various views of embedded systems

Register view

Firmware view

Instruction formats

Addressing modes