Introduction to Micro-processors

Dinesh Sharma

Department Of Electrical Engineering
Indian Institute Of Technology, Bombay

January 11, 2021

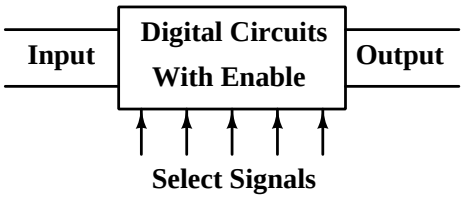
Need For a programmable device

- Design of complex integrated circuits has a high fixed cost component.
- If sold in large quantities, the cost of an integrated circuit can be quite low.
- It makes sense to design a programmable device which would perform different functions in different products.
- Now this device can be used for a variety of applications and will thus be required in large quantities.
- The function performed by the device will be selected by a digital input. This is called an **instruction**.
- The device is given a series of instructions to perform a function. This specific sequence of instructions is a **program**.

The Processor

- The programmable digital device which can run a program is called a processor.
- Processors can be large and complex. However, a (relatively) simple processor which is implemented on a single chip using VLSI technology is called a **microprocessor**.
- The term is applied these days even to complex processors – a pentium has upwards of 100 million transistors!
- The basic logic in a processor just performs the following loop endlessly:
  - Fetch next instruction
  - Decode it
  - Execute the instruction

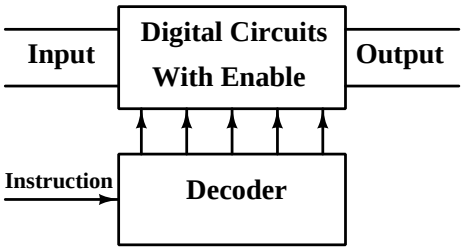
A multipurpose Digital Circuit



- Assume that we have multiple digital circuits, each performing a different function.
- A specific circuit can be selected using an enable input.

- We also need multiplexers to route the data from input to this circuit and to route its outputs to the overall output terminal.
- These multiplexers will also need select inputs.

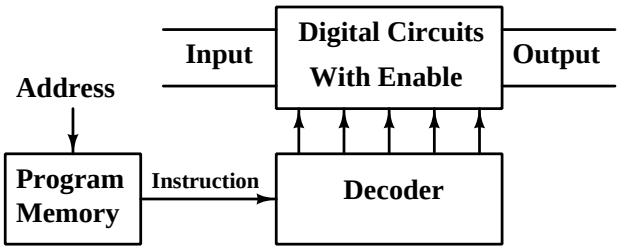
Choosing a specific sub-circuit



A lot of control inputs will be required to choose and configure a specific circuit and to route the data from input through this circuit to the output.

- We can encode this information in a compact form and use a decoder to generate all the control signals necessary for data routing and circuit selection.
- This encoded information is called an **instruction**.

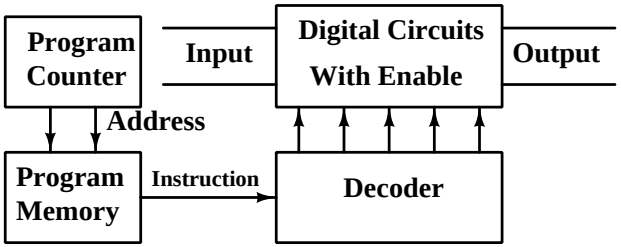
Storing Instructions



- The actual task may require a series of operations to be carried out.
- For example, we may wish to multiply a variable by a coefficient and add the product to an accumulating sum.

- We need a series of instructions, to be executed in a sequence.
- The sequence of instructions is called a **program**.
- It is convenient to store these instructions in a memory at sequential addresses.

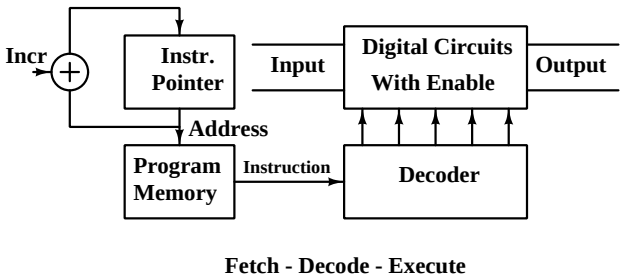
Fetching Instructions



- To fetch each instruction, we need to provide its address to the memory.
- From where will this address come?

- We can use a counter to generate sequential addresses.
- Every time an instruction is fetched from the memory, the counter will increment the address to point to the next instruction.
- The memory where instructions are stored is called the program memory.
- The counter which provides addresses to the program memory is called a program counter.

Fetching Instructions

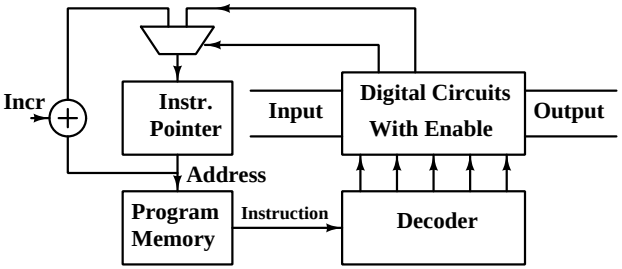


Fetch - Decode - Execute

- The program counter is just a register to store a value and an adder to increment the value stored in the register.
- This register is often referred to as the instruction pointer.

- The instruction pointer needs to be initialized to some value when the whole operation starts.
- The ability to load data into the instruction pointer gives us the ability to alter the flow of instructions in the program.
- When a new value is loaded in the instruction pointer, the next instruction will be fetched from this new location.

Fetching Instructions from non sequential address



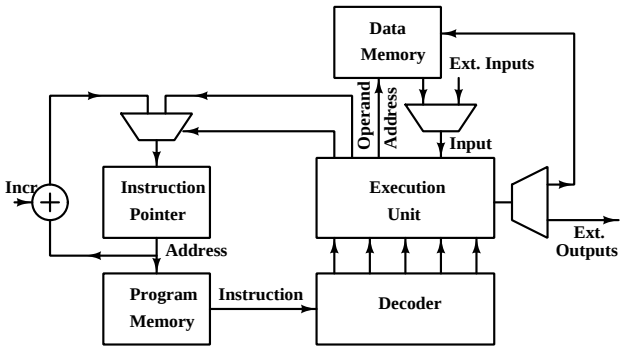
Fetch - Decode - Execute

- Instructions need not be executed in a strict sequence.
- After finishing a block of instructions, we may wish to execute a different block of instructions, stored at some other address.

- Moving to a different block of instructions may be accomplished by loading a different value into the instruction pointer.
- The value to be loaded into the instruction pointer can be provided by the last instruction of the first block.
- execution of this instruction will cause the multiplexer to choose this new address, rather than the default incremented address as the input to the instruction pointer.

Managing data

So far we have concentrated on the flow of instructions and control path. What about the data path?



Fetch - Decode - Execute

- Different instructions will, in general, operate on different data. How do we propose to manage the flow of data?
- Data can also be stored in a memory. Successive data items can then be fetched from/written to this memory.

- We shall need the address where the data item is to be fetched from/written to.
- The actual address of the data item is often stored in a register. (This register is called a data pointer).
- The instruction provides the address or the name of the register which stores the address.

Registers in the processor

- Where do we store intermediate results and addresses in this multipurpose digital circuit?
- Just like the instruction pointer and data pointer, the processor may contain other registers which are used for specific purposes.
- In addition, it may have general purpose registers which are used for storing intermediate data during computations.
- Registers which are specifically used to store addresses are called pointers. For example, the Instruction pointer, Data pointer.
- There is also a stack pointer – but more about that later . . .
- Registers whose contents can be manipulated through instructions are placed in a **register file** and are often identified by indices allotted to them.

Accessing data

- Data items to be operated on are called operands.
- An instruction may require multiple operands. For example, an ADD instruction uses two operands for adding and also needs information about where to store the result.
- The instruction should not only specify what operation to perform, but should also provide information about where the operands are and where to store the result.
- An instruction may specify operands in different ways. For example, the operand may be included in the instruction itself, or it may be in a local storage register etc.
- These are called Addressing modes.

Addressing Modes

Immediate operands: The value of an operand may be included in the instruction itself. This method is called immediate addressing.

For example, in 8051: MOV A, #27 means: place the value 27 in register A (Accumulator)

The two operands are the value 27 and the destination register A. The value is specified using immediate mode, while the destination is specified using register addressing (described next).

In 8051 programs, immediate values must be preceded by the hash symbol # .

Addressing Modes

Implied Immediate operands: Sometimes the immediate data may be implied and need not be included explicitly.

For example, the instruction to increment an operand asks for the number ‘1’ to be added to the operand.

Here, ‘1’ is an implied immediate operand for addition. (The operand to be incremented will have to be specified using other addressing modes, of course).

Similarly, the instruction CLR A means move the value ‘0’ into register A. Here the operand ‘0’ is implied.

Addressing Modes

Register operands: Most microprocessors include registers for local data storage. The instruction may specify that an operand is stored in a specific register.

For example, an 8051 provides registers A, B, and the eight registers R0-R7.
In the example used for immediate addressing:
MOV A, #27
the destination register was specified using register addressing.

Similarly, ADD A, R0
means add the value stored in register R0 to the accumulator (register A). Both operands use register addressing here.

Addressing Modes

Direct Addressing: The value of an operand may be stored in data memory. The instruction can include the **data memory address** from which the operand must be fetched. This is called Direct addressing.
Notice the difference – in immediate addressing, we place the **value** of the operand in the instruction. In direct addressing, the instruction includes the **address** from which the operand must be fetched from the data memory.

How do we distinguish whether a given number is the value or the address?
In 8051 software, a number is interpreted as an **address** unless it is preceded by a hash symbol: # .
A number following a hash sign (#) is interpreted as an immediate number.

Addressing Modes

Register Indirect Addressing: In this mode of addressing, the operand is in the data memory, but rather than including its absolute address in the instruction, we specify a register which contains the address.

Notice, that the register contains the **address** of the operand here, not the value of the operand.

In 8051 software, register indirect addressing is signaled by placing an @ sign before the register name. Assemblers for some other processors enclose the register name in square brackets to indicate indirect addressing.
MOV A, @ R1 will read the data memory location whose address is taken from R1, and place the value returned by the memory in register A.

Addressing Modes

Indexed Register Indirect mode: This mode uses two registers. One is used as the "base address" and the other as the index. Contents of these two registers are added and the sum provides the address where the operand is to be found.

8051 provides a 16 bit register called DPTR (data pointer). This register can be used as the base registered and the accumulator as the index.

MOVC A, @A + DPTR
Takes the current (16bit) value in DPTR and adds the (8 bit)contents of A to it. Because it is a MOVC (move constant) instruction, it goes to the **program memory** with this address and fetches an 8 bit value from there. This value is then stored in A.

Other Addressing Modes

Additional addressing modes: There are many other addressing modes. For example, an address register may be auto-incremented (or decremented) at every use. This mode is used for the stack pointer. (Discussion of stacks and stack pointer will be taken up later)

Other microprocessors provide many additional addressing modes.
For example, 8086 provides modes involving a base address register, a displacement register and an index register.

However, we shall not discuss these right now.

Some addressing modes are specific to control flow. These are used to generate addresses from where the next instruction will be fetched.

Addressing Modes for Control Flow

These modes are used for loading values in the instruction pointer.

direct jump: This mode provides a full address which is to be loaded into the instruction pointer.
LJMP Addr_{16}
loads the given 16 bit address into the instruction pointer. So the next instruction is executed from this address onward.

Relative addressing: This takes the current value of instruction pointer and adds the given (signed) displacement value to it. Thus the next instruction to be executed is fetched from a location displaced from the current location by the given amount. The usage is:

SJMP $\text{Signed Displacement}_8$

Addressing Modes for Control Flow

Hybrid jump call In this, a part of the instruction pointer is modified, while the most significant bits are retained. This results in a local jump restricted to a window around the current location.
AJMP Addr_{11}
will replace the least significant 11 bits of the instruction pointer with the given 11 bit address. The top 5 bits remain unchanged.

Jump to a variable address This functionality is provided by the JMP instruction of 8051.
JMP @A + DPTR
Loads the value obtained by adding the 16 bit contents of DPTR and 8 bit contents of the accumulator into the instruction pointer. This instruction provides a way to jump to a variable address depending on the current value in A or DPTR.

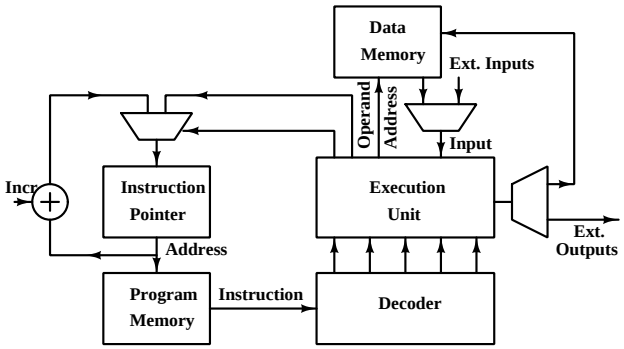
Addressing Modes and Programs

How do these addressing modes relate to objects in high level programming languages – for example, C?

C object	Addr. mode
Constant	Immediate
Intermediate variable	Register
variable	Direct
Array element	Indirect / indexed
Structures	Indirect /indexed

Data I/O with external world

The circuit we are designing should also be capable of fetching data from external sources and to output data to destinations other than the data memory.



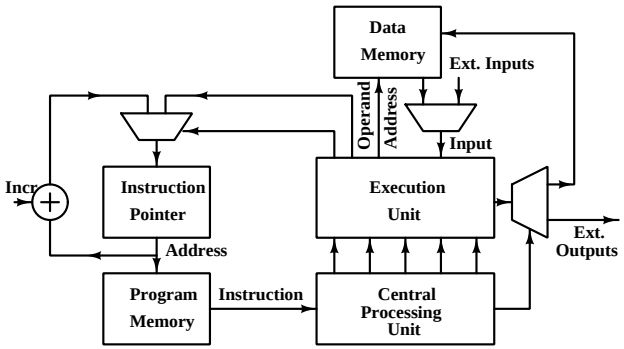
Fetch - Decode - Execute

- We can place multiplexers in the input data path and output data path to control the data path in order to choose between data memory and external IO.
- Control for these multiplexers should also be included in the instruction.

- The data and program memory need not be physically distinct.
- Different architectures choose independent paths for instructions and data or a shared path with a common memory for both instructions and data.

The Central Processing Unit

We have oversimplified the evolution of this circuit by assuming that just a combinational circuit like a decoder is needed to execute instructions.



Fetch - Decode - Execute

- In fact each instruction may need a sequence of actions to be executed properly.
- Therefore, we need a sequential circuit like a finite state machine to interpret and carry out an instruction.

- The circuit which generates the sequence of control inputs to the execution unit is called the Central Processing Unit or CPU.

The Central Processing Unit

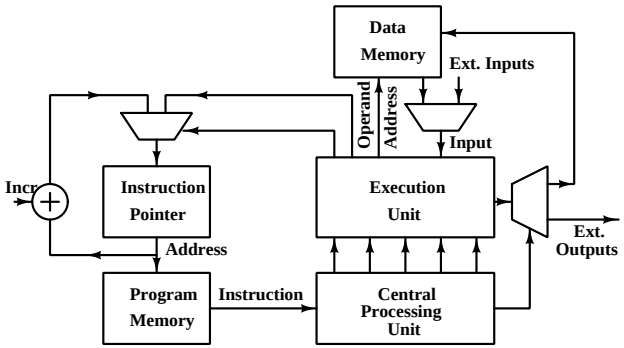
- The central processing unit generates control inputs for the sequence of operations to be carried out.
- At the top level, it repetitively carries out the three steps:
  - Fetch the next instruction
  - Decode the instruction
  - Execute it
- Each of these operations may involve a series of transactions with the memory or with internal execution units.
- Instruction execution may involve data processing or controlling the flow of instruction through writing to the instruction pointer.

Interrupting the processor

- While executing the program, some external events may occur which must be handled before we proceed with the execution of a problem.
- For example, a printing program may need to be interrupted if we run out of paper.
- A processor will need to provide the infrastructure such that when an interrupt request comes:
  - Execution of the current program is halted and a different program, which handles the external event, is run.
  - When this “handler” program gets over, the original program should start executing from exactly the point where it was halted.
- Results of the main program should be indistinguishable from the case where no interrupt had occurred.

So this is a microprocessor!

The general purpose digital circuit we have evolved is the basic core of a microprocessor.

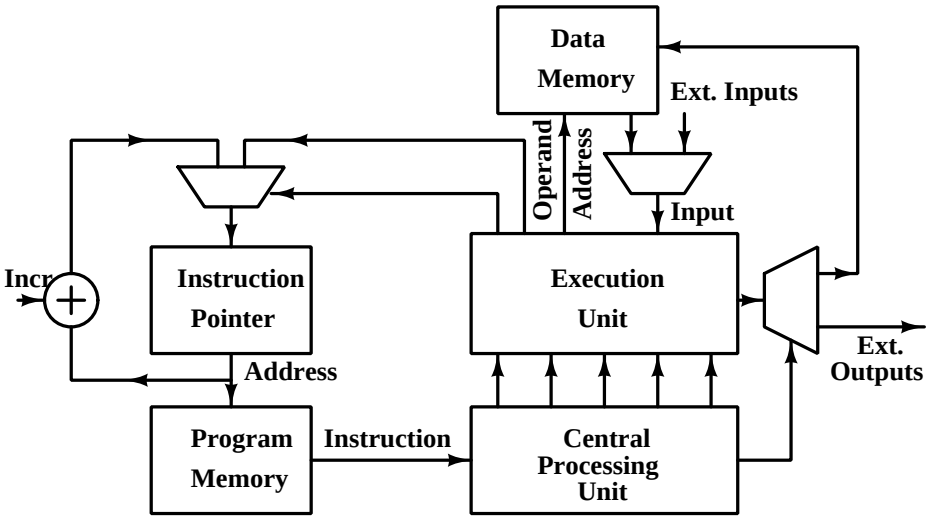


Fetch - Decode - Execute

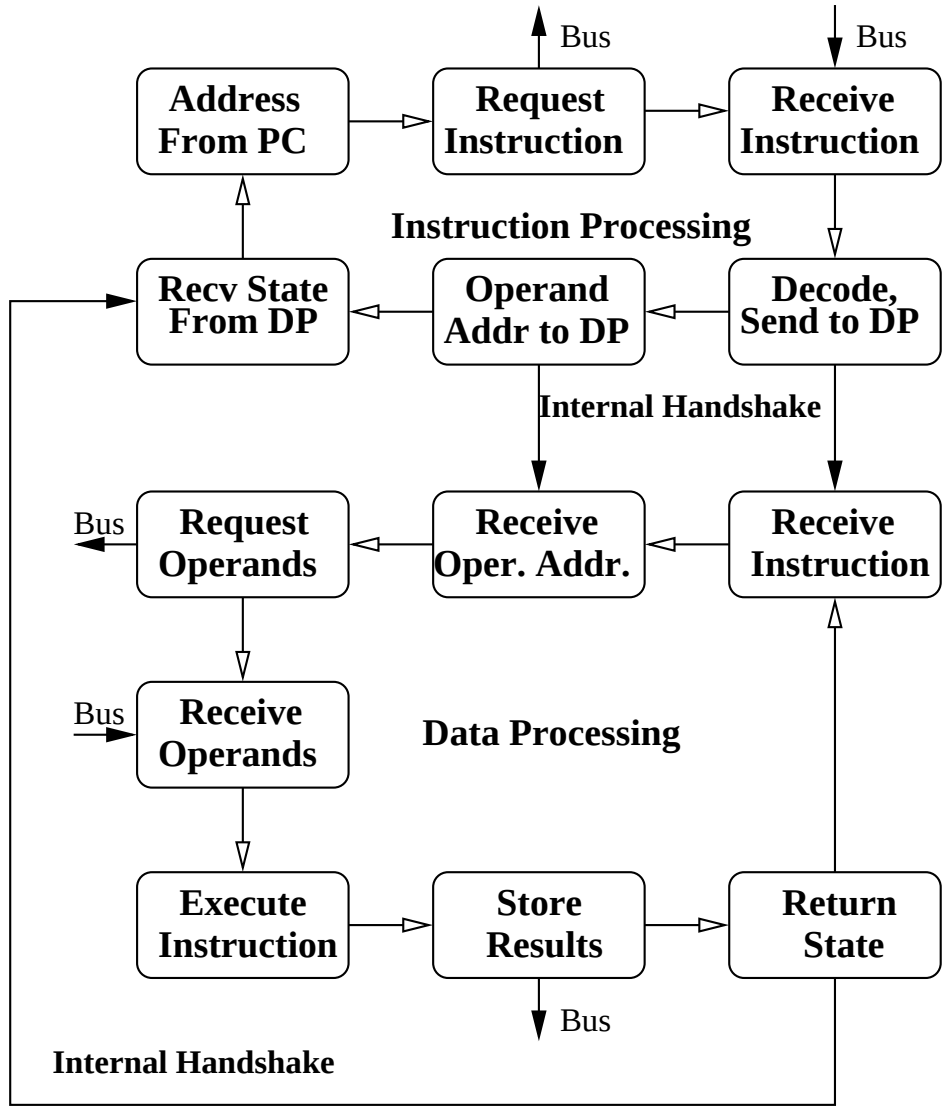
- A lot of details have to be worked out before we have a practical circuit which can be used for real applications.
- For example, what should be the width of the instruction path and data path?
- How many registers should we provide for efficient operation?

- How do we measure and optimize the performance of this circuit?
- These are the topics we shall discuss in this course.

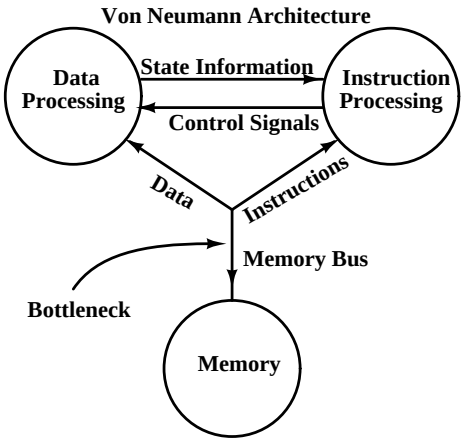
Instruction and Data Processing

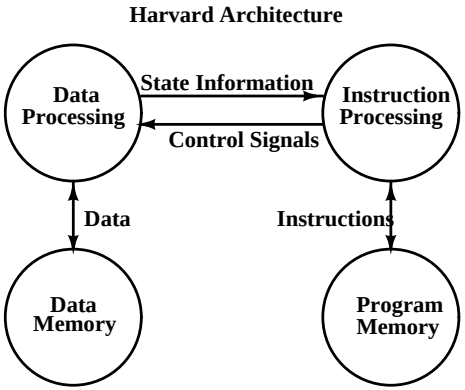


Fetch - Decode - Execute



Processor Architecture





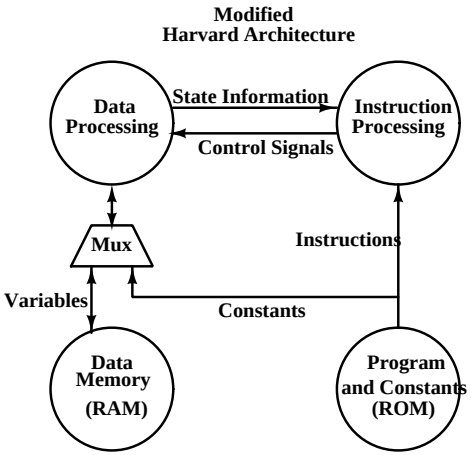
Von Neumann Architecture

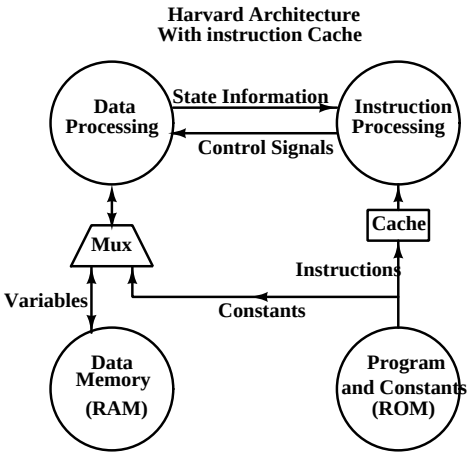
- In Von Neumann architecture, a common bus is used for data as well as instructions.
- The system can become ‘bus bound’.

Harvard Architecture

- In Harvard architecture, we provide separate data and instruction paths which can improve performance.
- However, it needs 2 buses $\rightarrow$ expensive!
- Traffic on the buses is not balanced.
- Instruction bus may remain idle.

Modified Harvard Architecture



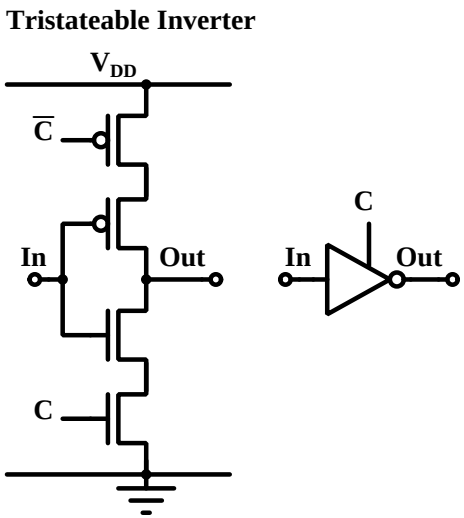


- Constants can also be stored with Instructions in ROM.
- Now, better Bus balancing is possible.
- For example, to compute $\sum C_i x_i$, we need one instruction read, 1 constant read, 1 data read and 1 result write.
- This gives 2 mem ops per bus.
- Adding a cache allows optimum utilization of bus bandwidths.
- Each operation need not be balanced individually.

THAT'S ALL FOR NOW

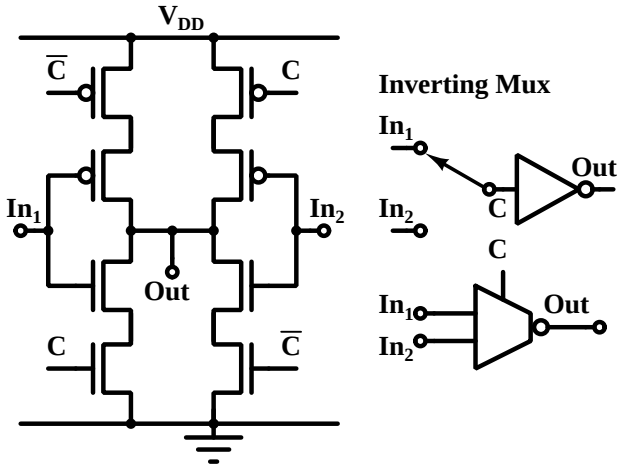
for generic architecture of microprocessors.

Multiplexers and demultiplexers for routing data



The figure on the left shows a tri-stateable inverter.

When C = '1', Out = $\overline{\text{In}}$.
When C = '0', Out is disconnected (Z state).

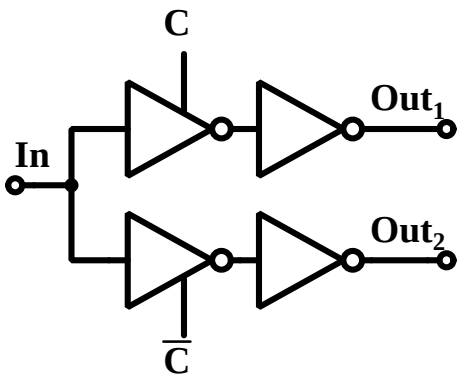


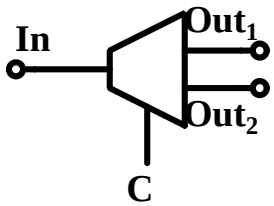
Using two tri-stateable inverters, we can make an inverting multiplexer.

When C = '1', Out = $\overline{\text{In}_1}$, and
When C = '0', Out = $\overline{\text{In}_2}$

Multiplexers and demultiplexers for routing data

- A multiplexer selects one out of many signals and puts it on a single line.
- A de-multiplexer carries out the reverse operation - It takes a digital signal from a single line and copies it to a selected line out of multiple lines.



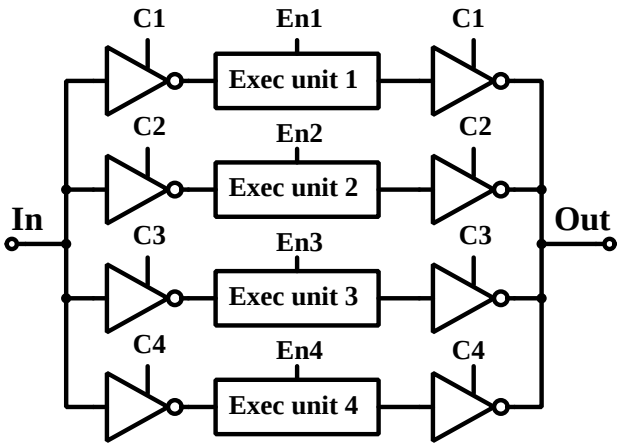


We can use two tri-stateable inverters with a common input enabled by complementary signals to implement a de-multiplexer.

We can add ordinary inverters if we do not want inversion, as shown in the figure here.

Multiplexers and demultiplexers for routing data

- Circuits shown in previous slides were ‘two way’ muxes and de-muxes.
- By adding a decoder and using its multiple outputs instead of C and $\overline{C}$, we can make a ‘2^n way’ mux or de-mux.



- The figure on the left shows an example of routing through an enabled circuit.
- In general, data paths will be multi-bit and therefore banks of demultiplexers/multiplexers will be required.

Back to ▸ a multi-purpose digital circuit