Computer Science 244

Notes: Theory Test 2

Rhys Williams

A message from our sponsors:
Do check out our test answers page, for all your semester test needs.

Definitions

Term	Definition
Absolute Path Name	Location of a filesystem object relative to the root directory.
Associative cache memory	A cache with n possible entries for each address is called an n-way set-associative cache. This is needed when our cache might try store different addresses in the same cache line due to a limited indexing system.
Belady's Anomaly	The more page frames does not necessarily decrease the number of page faults for certain memory access patterns (e.g FIFO gets worse)
Bus Skew	A problem where the signals on different lines of a bus travel at slightly different speeds, causing the signals to go out of time.
Copy-on-write	If multiple callers ask for resources which are initially indistinguishable, you can give them pointers to the same resource.
Dynamic Linking	When linking is postponed until execution time
Dynamic Loading	
Edge Triggered (flip-flops)	Circuit becomes active at either positive or negative edges of the clock signal.
External Fragmentation	After a system has been running for a while, memory is divided into "chunks". Some containing segments and others containing holes.
Internal Fragmentation	When a program does not fill an integral number of pages exactly and there is wasted space in memory.
Register Renaming	A form of pipelining that deals with data dependencies between instructions by remapping OS register requests to other, secret, registers - which prevents data corruption by separating our registers for different instructions happening at the same time.
Relocatable Code	Relocatable code resides at the address relative to the beginning of absolute code, when it is not known at compile time where a process is supposed to reside in memory. Thus, the code can run on any system and is guaranteed not to fight over memory/stack space, as the OS can assign the absolute code reference, from which relative references can be determined.
Sequential Access	A group of elements is accessed in a predetermined ordered sequence.
Sign extension	Duplicating the MBR (leftmost) sign bit into the upper 24-bit positions of the B bus.
Thrashing	Generating page faults frequently and continuously.

Translation lookaside buffer	Small, fast-lookup hardware cache. It is associative high-speed memory.
Unified Cache	Instructions and data use the same cache.
Volatile Memory	Storage that only maintains its data while the device is powered.
Write Allocation	Bringing data into cache on a write miss.
Write-through	Immediately updating an entry in main memory.

3.2. The Basic Digital Logic Level

1. Integrated Circuits

Instead of modular circuit design for small use, or even for the creation of modules for larger circuit use, integrated circuits (all-in-one on a single chip) are used. Arrays of components, microscopic circuits and semiconductor wafer material base are integrated together to form a single chip, which performs operations as directed.

IC's are mounted in packages (with the die - encompassing the actual computing unit - being in the middle). These packages can be seen as follows:

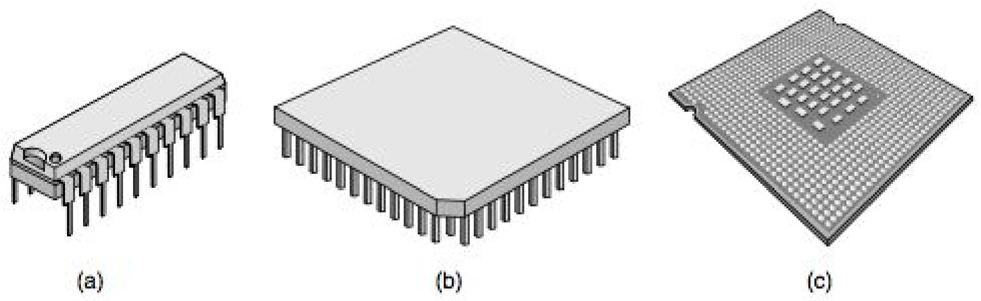
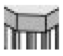


Figure 3-10. Common types of integrated-circuit packages, including a dual-in-line package (a), pin grid array (b), and land grid array (c).

Think of your typical CPU. They usually use a LGA (Land Grid Array) format as seen above in C. The wedges on the side of the packages (, ,) are there to stop morons like me putting the chips in the wrong way (I've done this often, so I thank the person who came up with that).

2. Combinational Circuits

Many applications of digital logic requires a circuit with multiple inputs and outputs, in which outputs are determined based on input values.

Multiplexer

2^n Inputs ($D_0 \ to \ D_{2^n-1}$)
1 Output
n control inputs

- Let D_x be the input x, where x is one of 2^n.
- x can be represented in binary.
- This binary representation is then placed into $N_0, N_1 ... N_n$
- This then selects a single input corresponding to the binary control input.

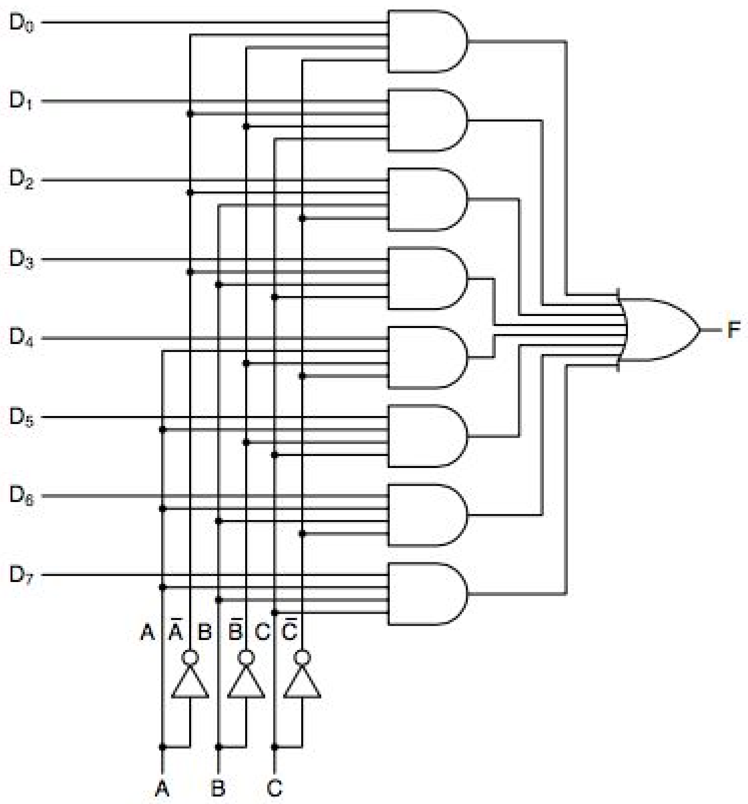


Figure 3-11. An eight-input multiplexer circuit.

Decoder

n inputs
2^n outputs

- Let D_x be our outputs, where x is one of 2^n.
- Let us represent x in terms of its binary encoding
- This binary representation is then placed into $N_0, N_1 ... N_n$ etc

- This selects a single output to push 'HIGH', hence outputting a '1' on the output D_x corresponding with our binary representation of x.

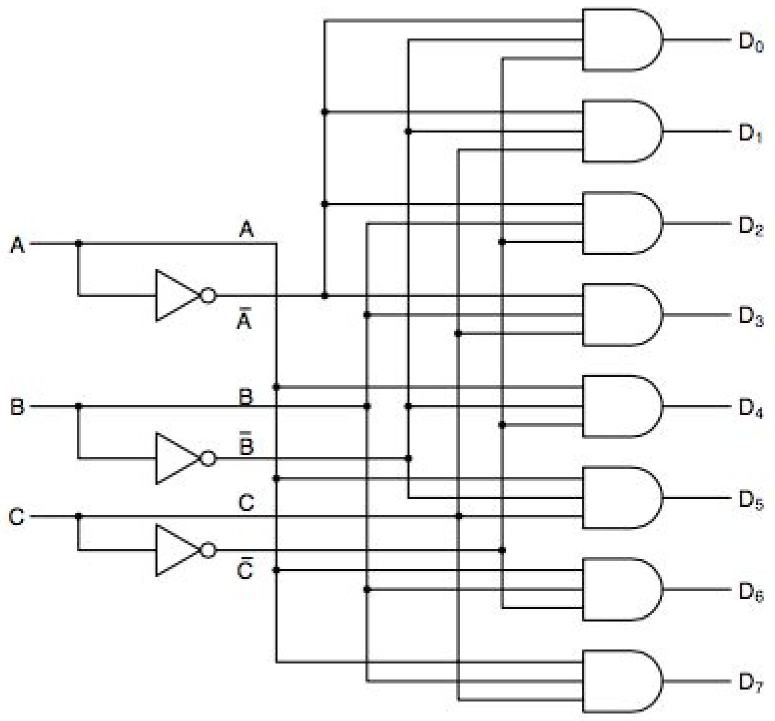


Figure 3-13. A three-to-eight decoder circuit.

Comparator

n inputs of length l
1 output ('HIGH' if inputs are equal, low if not)

The circuit thus compares each bit ($n_{1,0} = n_{2,0}\ and\ n_{1,1} = n_{2,1}\ etc\ etc$) to determine if the inputs are equal. Only if all inputs are equal, will the output be put 'HIGH'

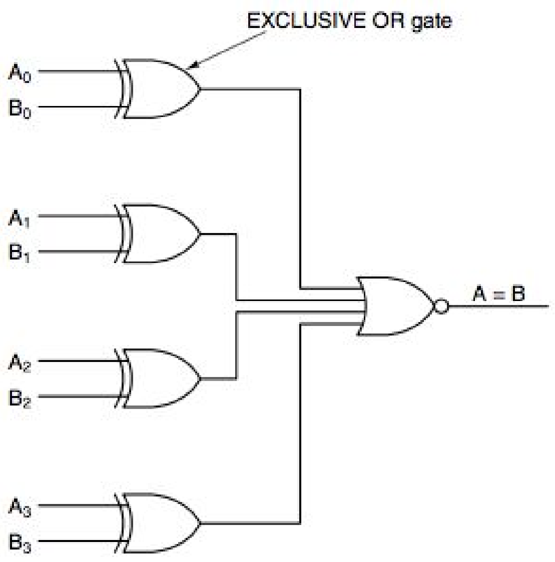


Figure 3-14. A simple 4-bit comparator.

3. Arithmetic Circuits

Shifter

The shifter performs a right or left arithmetic shift depending on the provided control bit C being set 'HIGH' or 'LOW'. Each input Dx is connected one bit left as well as one bit right, and it's path is thus chosen depending on the value of C.

If C is 'HIGH', the right and-gate is activated, thus shifting the bit right.
This is true too for when C is 'LOW', for which the left and-gate is activated.

On a left shift, 0 is inserted into D7. On a right shift, 1 is inserted into D0.

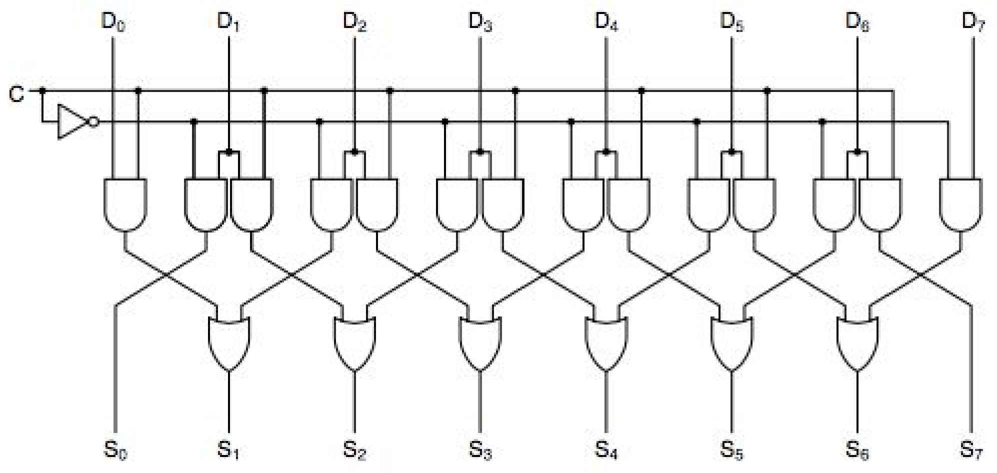


Figure 3-15. A 1-bit left/right shifter.

Adders

The below adders are used for 1-bit operations. We join the full-adder below in groups for computing larger bit-sizes. For example, we will link 16 full-adders to create a 16-bit adder.

Half-Adder

A half-adder computes a sum for inputs A or B. In order to create a complete adder, at least two of these must be combined, as they cannot themselves compute more than a single bit sequence. This is because they do not take any carry in from an external source, which means that when these are being computed in a combined fashion, all carry bits will be lost (thus A=1, B=1, A+B will be the same result as A=0, B=0).

The truth table and circuit for a half-adder looks as follows.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

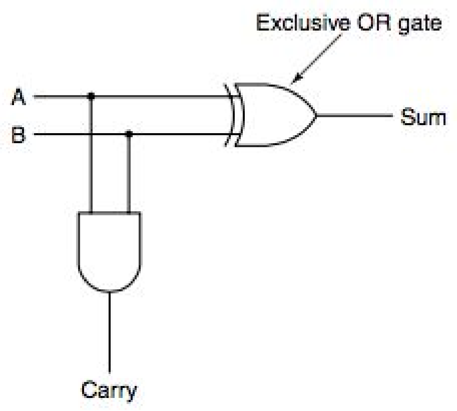


Figure 3-16. (a) Truth table for 1-bit addition. (b) A circuit for a half adder.

Example:
https://upload.wikimedia.org/wikipedia/commons/thumb/9/92/Halfadder.gif/220px-Halfadder.gif

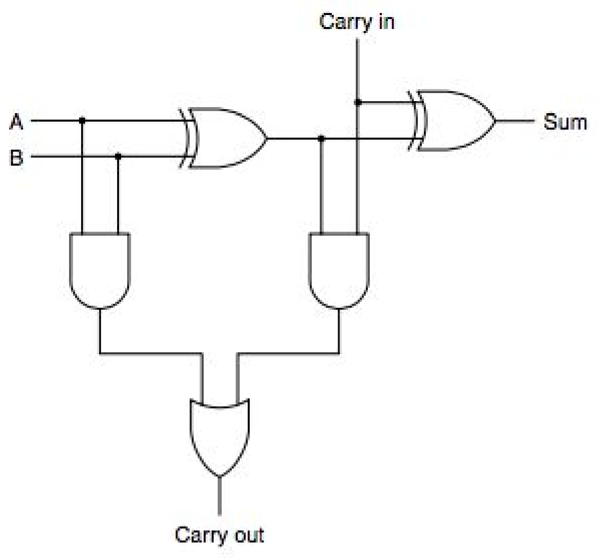
Full-Adder

A full adder improves on our half-adder by combining two half-adders and then allowing an external carry-in to enable us to combine them (for computing 16 bits, 32 bits etc).

A full adder is what we use when expanding to our larger bit-size systems.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Figure 3-17. (a) Truth table for full adder. (b) Circuit for a full adder.

Example:
https://upload.wikimedia.org/wikipedia/commons/thumb/5/57/Fulladder.gif/220px-Fulladder.gif

Speed-Up 'Carry Select Adder'

The below adders are used for 1-bit operations. We join the full-adder below in groups for computing larger bit-sizes. For example, we will link 16 full-adders to create a 16-bit adder.

To speed up our computation, we can implement a 'Carry Select Adder'. This splits up, for example, a 32 bit adder into an upper and lower 16-bit adder.

If we duplicate the upper-16 bit adder, we can thus create a system whereby we solve our solution in parallel, thereby halving our execution time. To do this, we execute our 1x lower and 2x upper adders at the same time.

1x Lower Adder receives the input carry bit:
    Computes in parallel

1x Upper Adder receives a carry bit of 1:
    Computes in parallel

1x Upper Adder receives a carry bit of 0:
    Computes in parallel

Now, depending on the final carry-bit of the lower adder, we merely select the correct upper-adder solution. This effectively 'multi-threads' our addition, and halves its execution time.

Arithmetic Logic Unit (ALU)

Most computers contain a single circuit for performing the AND, OR and SUM of two machine words.

Depending on an instruction input provided in F0 and F1, the ALU performs the requested operation - which is one of four operations.

- A and B
- A or B
- Not B
- A + B

The ALU also accounts for a carry-out and carry-in, and can thus be chained. Hence, a 16-bit ALU can be formed from 16 one-bit ALUs, etc.

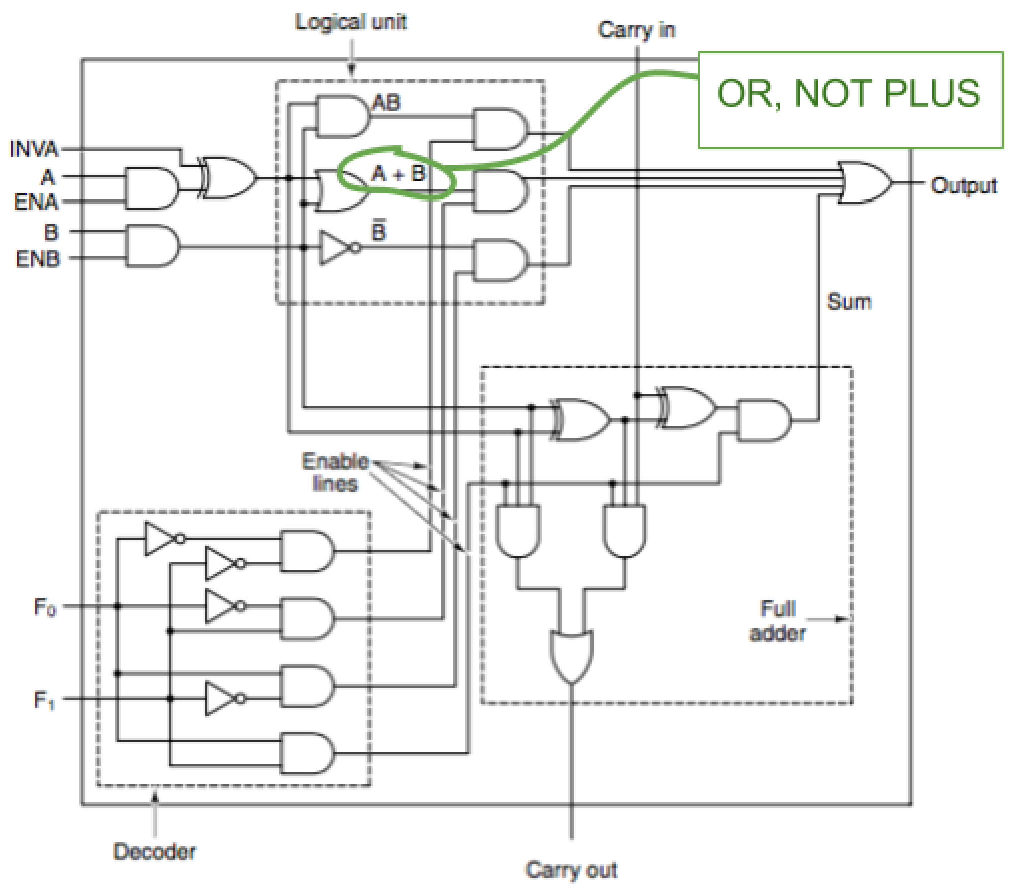
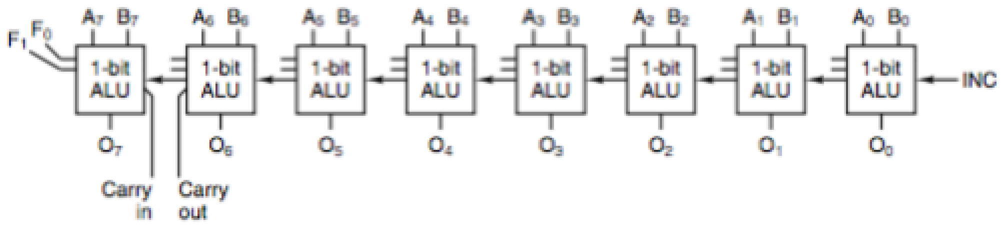


Figure 3-18. A 1-bit ALU.

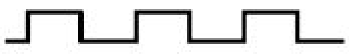


Instructions represented by F0 and F1 combinations:

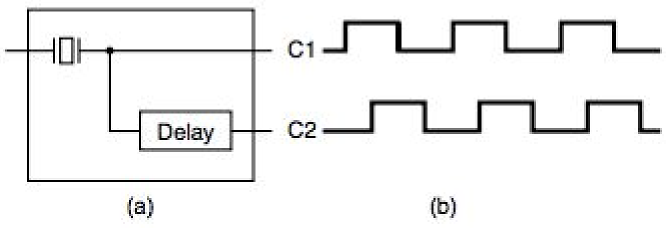
	F0	F1
A AND B	0	0
A OR B	0	1
B COMPLEMENT	1	0
A + B	1	1

4. Clocks

In order to schedule concurrent CPU/Arithmetic events, a clock is used. This clock provides timing through a 'HIGH', 'LOW' cycle, as seen below:



Sometimes, however, many events may happen during a single clock cycle. If these events must occur in a specific order, the clock cycle must be divided into sub- cycles. A common way of providing finer resolution than the basic clock is to tap the primary clock line and insert a circuit with a known delay in it, thus generating a secondary clock signal that is phase-shifted from the primary.

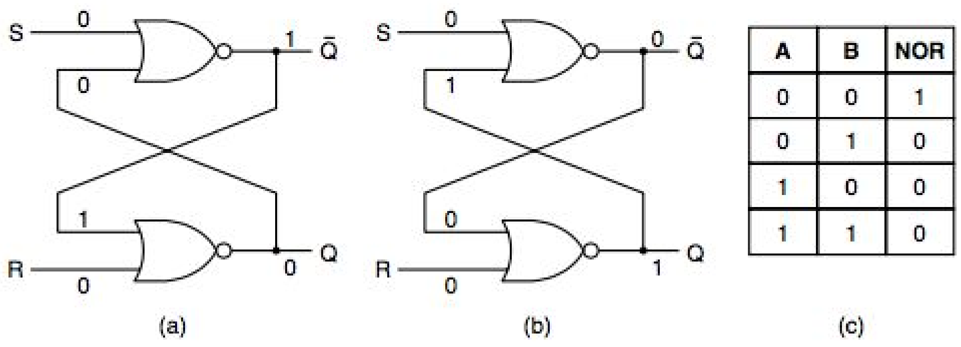


The clock is important and will be used throughout our circuit design process.

3.3. Memory

1. Latches

SR Latch



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Figure 3-21. (a) NOR latch in state 0. (b) NOR latch in state 1. (c) Truth table for NOR.

A latch is used for storing values for future use. Each individual latch can store a single bit until it is either reset, or given a new value to store. The above is a **SR Latch**, and contains:

S Input: Set the Latch
R Input: Reset the Latch

Have a look at (a) above. This latch is currently storing a value of 0 (which was provided by our latch being reset). As you can see, Q' is connected to our bottom NOR gate of Q. This thus creates a loop in which 0 is stored at Q.

Now, let's pulse S with a 1. Thus then flips our top NOR gate, thereby setting Q' to 0. This then pushes 0 to our bottom NOR gate, which therefore sets Q to 1. This now stores the last provided S value, 1.

Should we now input 1 into R, this will toggle our bottom NOR gate, which will then set Q back to 0. Note that R only acts on our latch when Q has a 'HIGH' value.

Clocked SR Latch

A clocked SR latch only enables our inputs S,R when our clock is at a 'HIGH'. This means that we can only modify our stored value on each clock cycle.

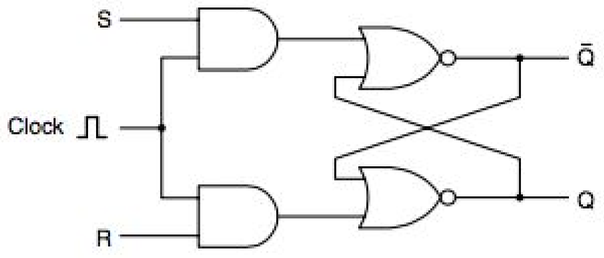


Figure 3-22. A clocked SR latch.

Note how if our clock (CK) is high, our AND gates become irrelevant. If CK is 'LOW', our inputs become irrelevant.

Clocked D Latch

Our clocked SR latch provides a problem when S = R = 1. Thus, to solve this, we remove one of the variables and introduce our clock to design a circuit which merely stores the last value provided at D. This is constantly updated at every clock cycle, and thus the last value of D at clock 'HIGH' will be stored.

This is known as true one-bit memory, as D=1 will make Q=1 until D=0 again. Thus, Q = D's last value at clock 'HIGH'.

It is important to note that this latch is level triggered, and thus will only be operational when our clock is 'HIGH'.

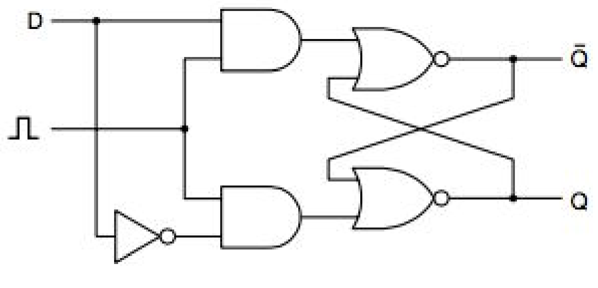
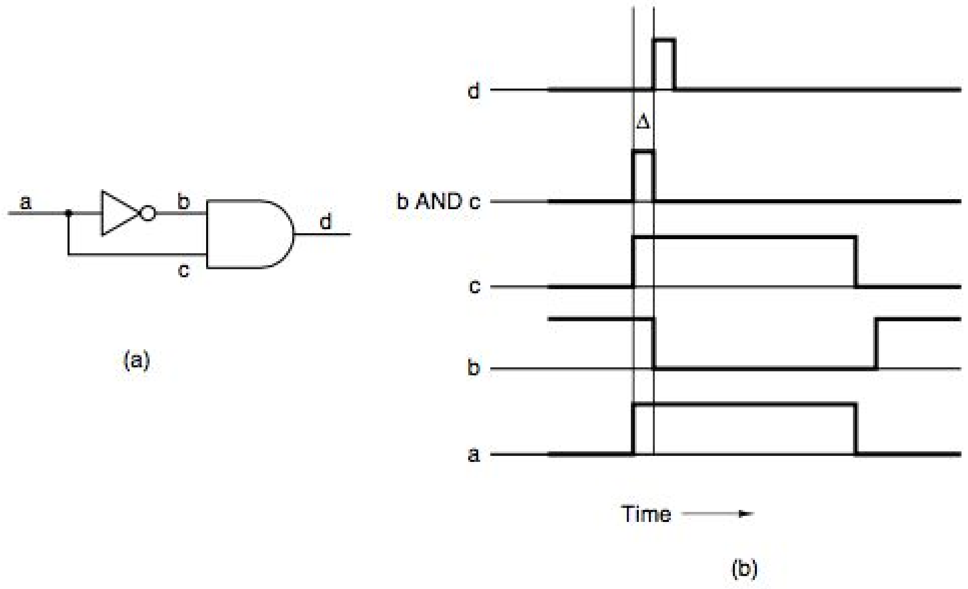


Figure 3-23. A clocked D latch.

2. Flip-Flop

As opposed to our latch being level-triggered, our flip flop is edge-triggered. This thus means that it is only activated when our clock is in transition phase (either from 'HIGH'->'LOW' or 'LOW'->'HIGH'). This means that the length of our clock pulse is irrelevant, and our value at D will be read only during a very quick transition period.



As seen above, we can create a circuit to provide a very quick pulse to our clock input for our latch when a transition occurs, thus changing our latch into a flip flop.

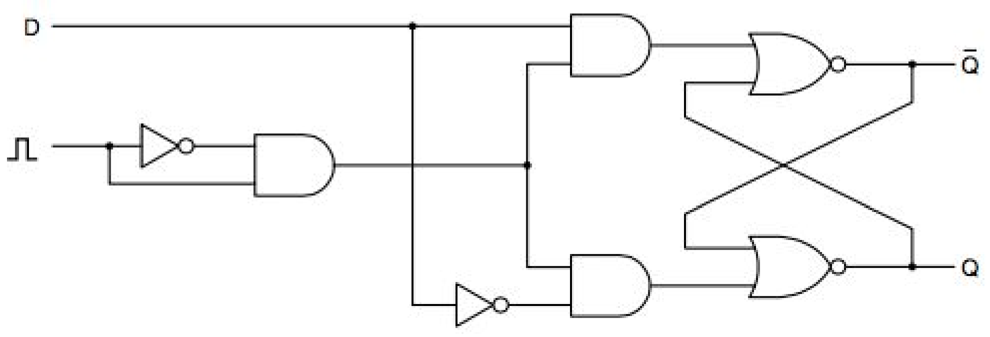


Figure 3-25. A D flip-flop.

Thus, the completed circuit holds above.

3. Registers

Flip-Flops can be combined to create familiar registers, which hold data types larger than 1 bit in length.

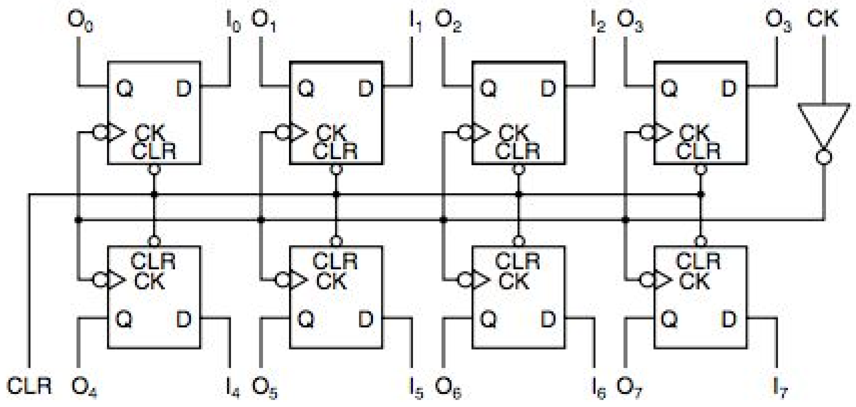


Figure 3-27. An 8-bit register constructed from single-bit flip-flops.

The above represents an 8-bit register. These can be combined in the same way (extending CLR and CK between groups) to create 16-bit, 32-bit registers etc.

Note the CK being inverted twice. This is purely to amplify the signal, and does not change the inner-workings of the circuit.

4. Memory Organization

We will not be required to draw the following diagram, however we must be able to understand and explain it.

To store values in memory, we do not need as complex a system as a register. Instead, we create a 12-bit chunk (which itself contains 4 x 3-bit words), with a requirement of only 13 signals (12-bit) as opposed to 20 for an (8-bit) register.

Each read/write operation always reads or writes a complete 3-bit word.

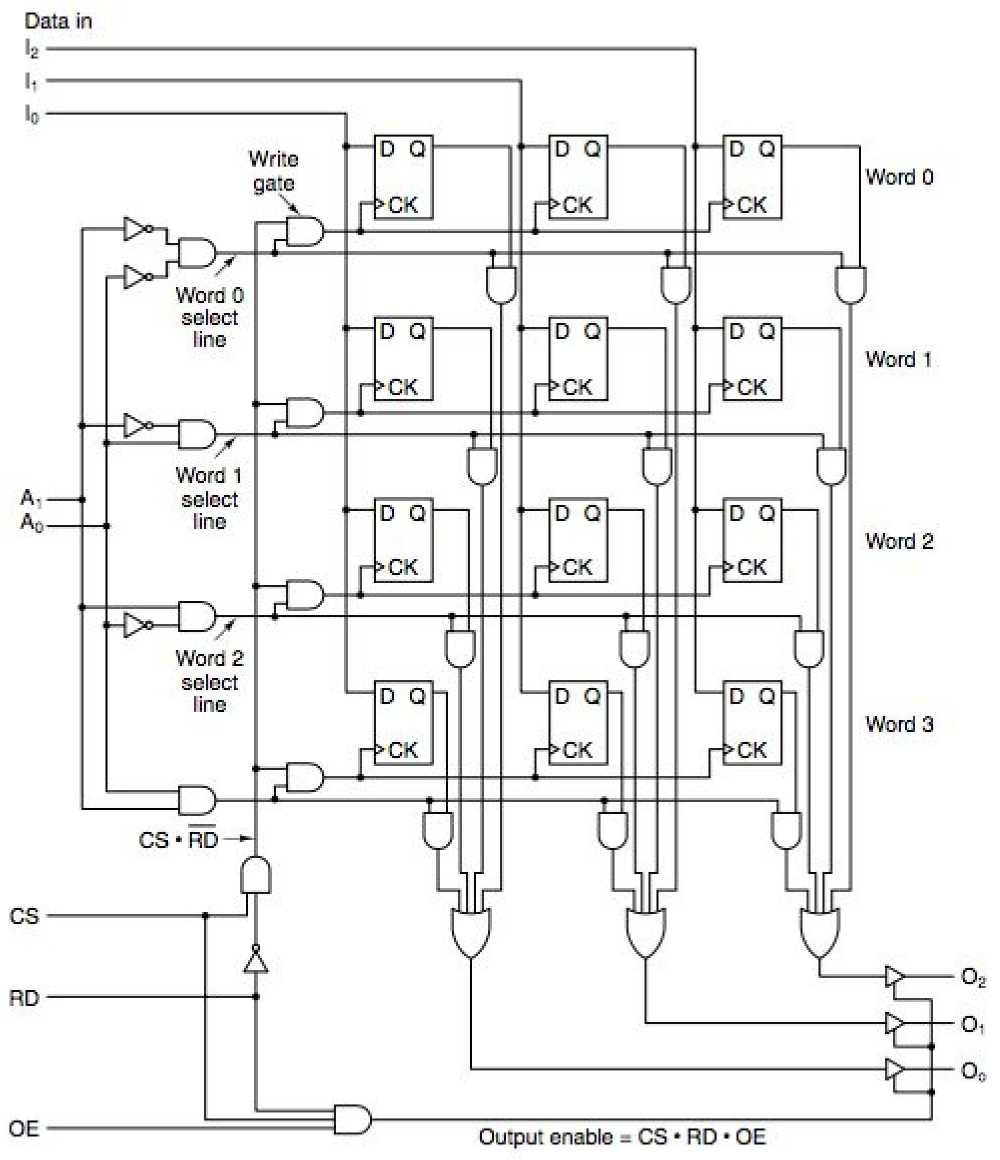


Figure 3-28. Logic diagram for a 4 × 3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

Important for the Diagram:
Input Lines: I0, I1, I2
Output Lines: O0, O1, O2
Address Lines: A0, A1
Chip Select: CS
Read/Write Select: RD
Output Enable: OE

To Read:
Set RD to 1
Set CS to 1
Set OE to 1
Select word through A0,1
Values in Word are sent to Output

To Write:
Set RD to 0
Set CS to 1
Set OE to 0
Select word through A0,1
Input in I0,I1,I2 is placed into Word

5. Memory Chips

The nice thing about the memory is that it extends easily to larger sizes. As we drew it above, the memory is 4 × 3, that is, four words of 3 bits each. To extend it to 4 × 8 we need only add five more columns of four flip-flops each, as well as five more input lines and five more output lines. To go from 4 × 3 to 8 × 3 we must add four more rows of three flip-flops each, as well as an address line A2. With this kind of structure, the number of words in the memory should be a power of 2 for maximum efficiency, but the number of bits in a word can be anything.

Because integrated-circuit technology is well suited to making chips whose internal structure is a repetitive two-dimensional pattern, memory chips are an ideal application for it. As the technology improves, the number of bits that can be put on a chip keeps increasing, typically by a factor of two every 18 months (Moore's law). The larger chips do not always render the smaller ones obsolete due to different trade-offs in capacity, speed, power, price, and interfacing convenience. Commonly, the largest chips currently available sell at a premium and thus are more expensive per bit than older, smaller ones.

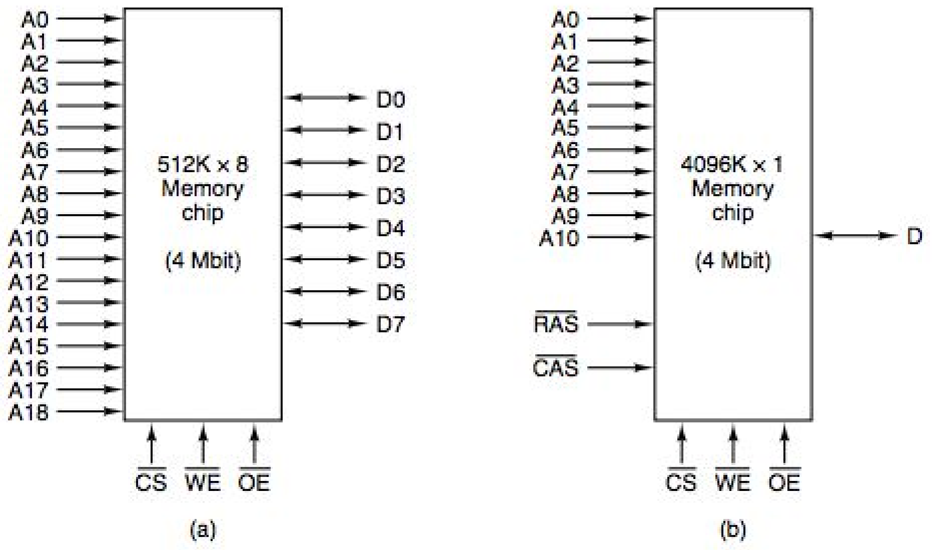


Figure 3-30. Two ways of organizing a 4-Mbit memory chip.

6. RAMs and ROMs

RAM (Volatile)
Random Access Memory. Can be Read/Written to.

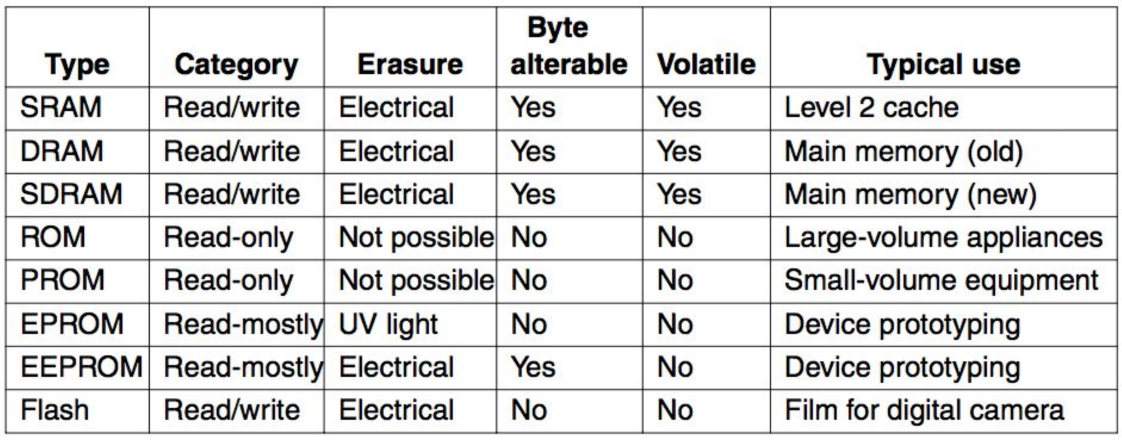
SRAM: Static Ram. Very Fast, but Expensive. Used for Cache on CPUs.
Are constructed internally using circuits similar to our basic D flip-flop.

DRAM: Dynamic RAM. Fast, Cheap. Used for Primary Memory on Most Computers.
Do not use flip-flops. Instead, a dynamic RAM is an array of cells, each cell containing one transistor and a tiny capacitor. The capacitors can be charged or discharged, allowing 0s and 1s to be stored. Because the electric charge tends to leak out, each bit in a dynamic RAM must be refreshed (reloaded) every few milliseconds to prevent the data from leaking away.

ROM (Non-Volatile)
Original ROMs (Read-Only Memory) could not be changed or erased, intentionally or otherwise. The data in the ROM was inserted during its manufacture, essentially by exposing a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface.

The only way to change the program in the ROM was to replace the entire chip.

See the table below for different technologies.



Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory (old)
SDRAM	Read/write	Electrical	Yes	Yes	Main memory (new)
ROM	Read-only	Not possible	No	No	Large-volume appliances
PROM	Read-only	Not possible	No	No	Small-volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

Figure 3-32. A comparison of various memory types.

3.4. CPU Chips and Buses

1. CPU Chips

All modern CPUs are contained on a single chip. This makes their interaction with the rest of the system well defined. Each CPU chip has a set of pins, through which all its communication with the outside world must take place. Some pins output signals from the CPU to the outside world; others accept signals from the outside world; some can do both. By understanding the function of all the pins, we can learn how the CPU interacts with the memory and I/O devices at the digital logic level.

It's important to understand that the CPU communicates with the memory and I/O devices by presenting signals on its pins and accepting signals on its pins. No other communication is possible.

Pins are divided into three types:

1. Address
2. Data
3. Control

Address Pins:

- Contains the memory address of the instruction.
- A chip with m address pins can address up to 2^m memory locations.

Data Pins:

- Contains data being carried to/from the CPU
- A chip with n data pins can read or write an n-bit word in a single operation. A CPU with 8 data pins will take four operations to read a 32-bit word, whereas one with 32 data pins can do the same job in one operation.

Control Pins:

- Bus control
  The bus control pins are mostly outputs from the CPU to the bus (thus inputs to the memory and I/O chips) telling whether the CPU wants to read or write memory or do something else.

- Interrupts
  The interrupt pins are inputs from I/O devices to the CPU. In most systems, the CPU can tell an I/O device to start an operation and then go off and do some other activity, while the I/O device is doing its work. When the I/O device is done, it asserts a signal on these pins to get the attention of the CPU.

- Bus arbitration
  The bus arbitration pins are needed to regulate traffic on the bus, in order to prevent two devices from trying to use it at the same time. For arbitration purposes, the CPU counts as a device and has to request the bus like any other device.

- Coprocessor signaling
  Some CPU chips are designed to operate with coprocessors such as floating-point chips, but sometimes graphics or other chips as well. To facilitate communication between CPU and coprocessor, special pins are provided for making and granting various requests.

- Status & Miscellaneous
  There are various miscellaneous pins that some CPUs have. Some of these provide or accept status information, others are useful for debugging or resetting the computer, and still others are present to assure compatibility with older I/O chips

2. Computer Buses

Definition

A bus is a common electrical pathway between multiple devices. They are categorized by their function.

Characterization:

- Internal:
  Connect CPU to ALU for Data Transfer

- External
  Connect CPU to Memory or I/O

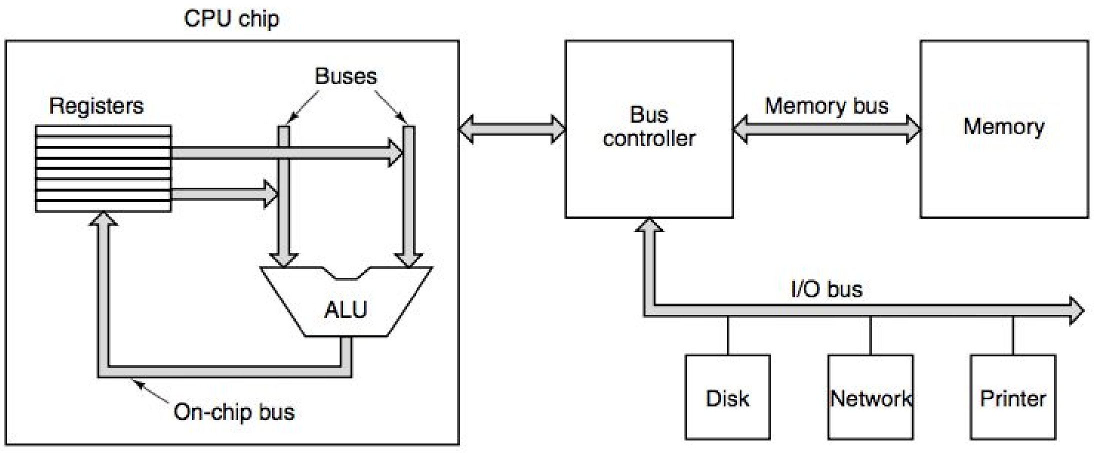
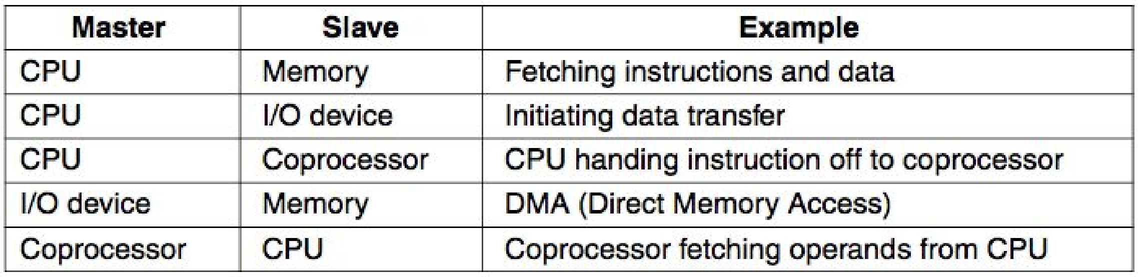


Figure 3-35. A computer system with multiple buses.

Bus Protocol

Buses each have a set of rules for communication. Think South African Road Rules, but for a highway with cars going at 10's of millions of KM per hour and where people actually stop at the stop streets. These are important for ensuring data can be transferred across the bus.

Master Devices are active, and are thus able to initiate bus transfers.
Slave Devices are passive, and wait for requests to act on.



Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O device	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Figure 3-36. Examples of bus masters and slaves.

The binary signals that computer devices output are frequently too weak to power a bus, especially if it is relatively long or has many devices on it. For this reason, most bus masters are connected to the bus by circuitry called a bus driver, which is essentially a digital amplifier. Similarly, most slaves are connected to the bus by a bus receiver. For devices that can act as both master and slave, a combined circuit called a bus transceiver is used. These bus interfaces are often tri-state devices, to allow them to disconnect when they are not needed.

To connect protocols together, decoder circuits might be needed to ensure that each device receives the input they need. For example, some CPUs have three pins that encode whether the CPU is doing a memory read, memory write, I/O read, I/O write, or some other operation. A typical bus might have one line for memory read, a second for memory write, a third for I/O read, a fourth for I/O write, and so on. The decoder would then match the signals up.

3. Bus Width

Bus width is the most obvious design parameter. The more address lines a bus has, the more memory the CPU can address directly. If a bus has n address lines, then a CPU can use it to address 2^n different memory locations. To allow large memories, buses need many address lines. That sounds simple enough.

However, this ultimately means that wider buses need more wires than narrow ones. These are both expensive and take up physical space on the motherboard (as well as need larger connectors).

Growth of Address-Lines:

When technology improves, this creates a problem:

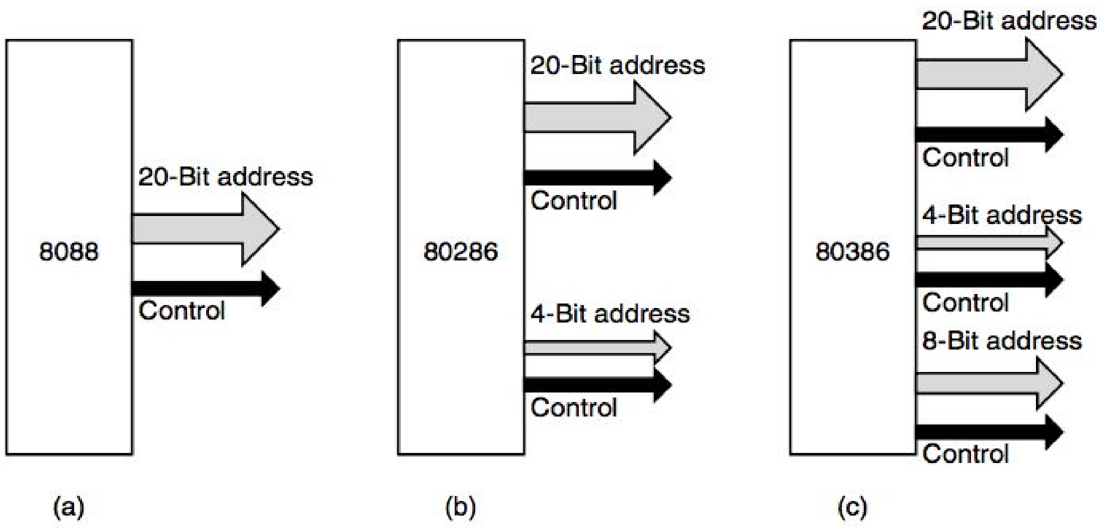


Figure 3-37. Growth of an address bus over time.

As you can see, after generational updates, the resulting design (c) is much messier than it would have been had the bus been given 32 lines at the start.

Growth of Data Lines:

Not only does the number of address lines tend to grow over time, but so does the number of data lines, albeit for a different reason. There are two ways to increase the bandwidth of a bus: decrease the bus cycle time (more transfers/sec) or increase the data bus width (more bits/transfer). **Speeding the bus up is possible (but difficult) because the signals on different lines travel at slightly different speeds, a problem known as bus skew**. The faster the bus, the more the skew.

Another problem with speeding up the bus is it will not be backward compatible. Old boards designed for the slower bus will not work with the new one. Therefore the usual approach to improving performance is to add more data lines/ As you might expect, however, this incremental growth does not lead to a clean design in the end.

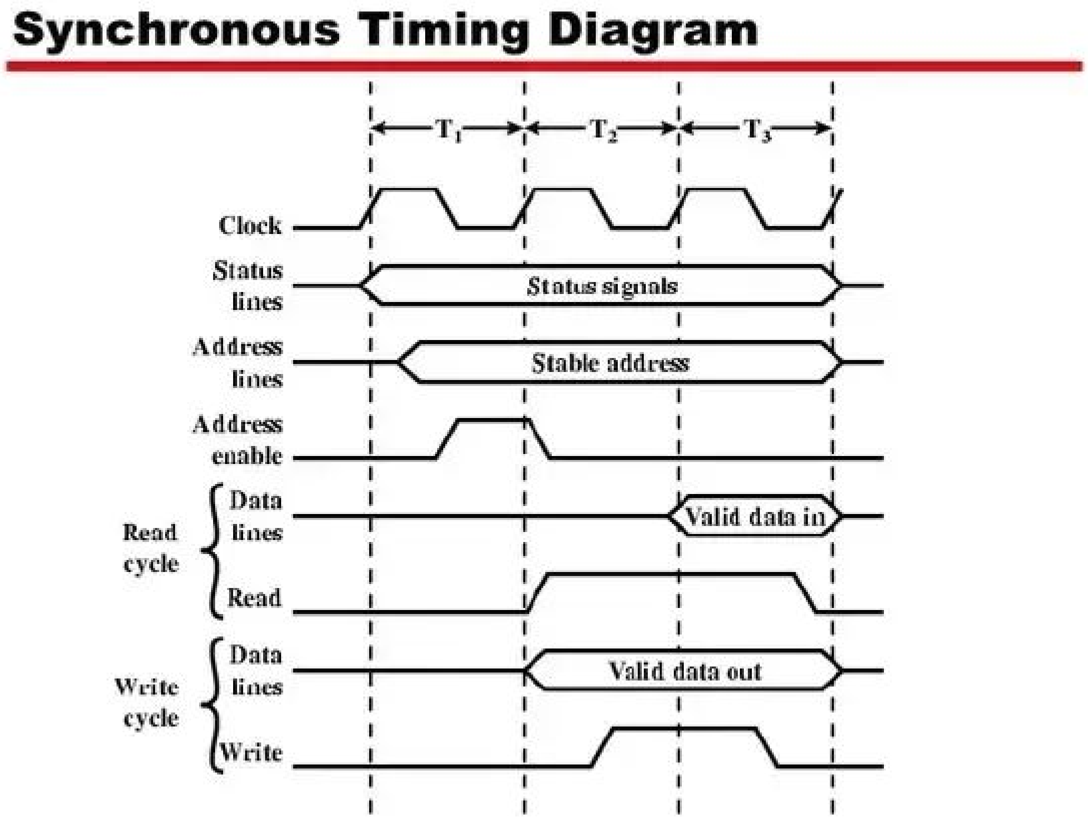
Another solution is a multiplexed bus. In this design, instead of the address and data lines being separate, there are, say, 32 lines for address and data together. At the start of a bus operation, the lines are used for the address. Later on, they are used for data. For a write to memory, for example, this means that the address lines must be set up and propagated to the memory before the data can be put on the bus. With separate lines, the address and data can be put on together. Multiplexing the lines reduces bus width (and cost) but results in a slower system.

4. Bus Clocking

Synchronous Buses

A synchronous bus transmits bits of data by using a shared clock signal to synchronize the timing of both the sender and the receiver. That's why it's called “synchronous".

1. Transmitter and receivers are synchronized of clock.
2. Data bits are transmitted with synchronization of clock.
3. Character is received at constant Rate.
4. Data transfer takes place in block.
5. Start and stop bit are required to establish communication of each character.
6. Used in high – speed transmission.



Asynchronous Buses

A asynchronous bus doesn’t have a master clock. All parties on an asynchronous bus use their own clocks to process incoming/outgoing data.

1. Transmitters and receivers are not synchronized by clock.
2. Bit’s of data are transmitted at constant rate.
3. Character may arrive at any rate at receiver.

4. Data transfer is character oriented.
5. Start and stop bits are required to establish communication of each character.
6. Used in low – speed transmission.

Asynchronous clocks are theoretically advantageous. However, most buses are synchronous as it is easier to implement with memory and the cpu. Such is life.

5. Bus Arbitration

Up until this point, we have assumed that each bus has a single master issuing commands. In reality, this is far from true. I/O and co-processor chips have to become bus master which issue commands to read and write memory, as well as cause interrupts.

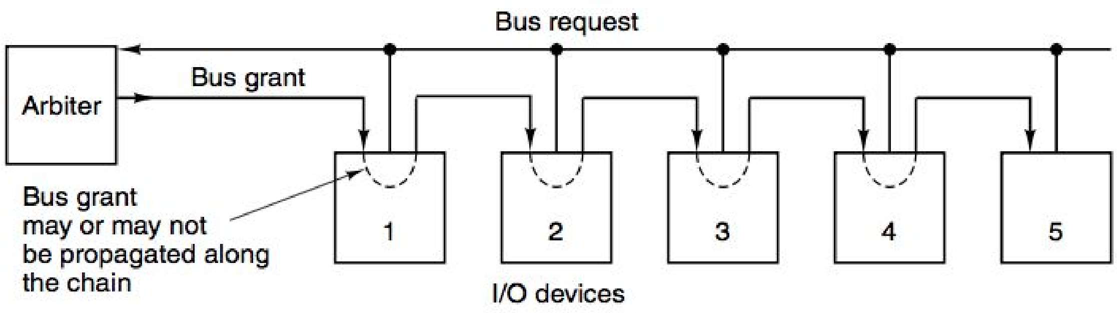
Think South African Road Rules, but each person standing on the side of the road gets a big red button which instantaneously switches a traffic light from green to red, and visa versa. If people kept pushing the button without any control, eventually you'll end up with chaos, where cars wouldn't know whether the light was actually green - or if it's green for only a split second. Hence, you need something to let the people know whether they are allowed to push the button or not.

In order to manage these requests, some bus arbitration (law maker) mechanism is needed to prevent chaos.

Daisy-Chaining (Centralised)

When the arbiter sees a bus request, it issues a 'you can be a master now' by asserting the bus grant (giving permission) line. This line is wired through all the I/O devices in series, like a cheap string of Christmas tree lamps. When the device physically closest to the arbiter sees the grant, it checks to see if it has made a request. If so, it takes over the bus but does not propagate the grant further down the line. If it has not made a request, it propagates the grant to the next device in line, which behaves the same way, and so on until some device accepts the grant and takes the bus. This scheme is called daisy chaining.

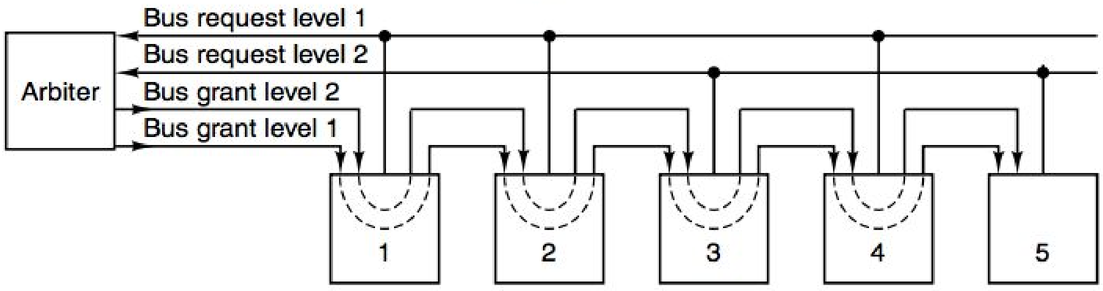
It has the property that devices are effectively assigned priorities depending on how close to the arbiter they are. The closest device wins. A centralized one-level bus arbiter using daisy chaining



In this scheme, a single bus arbiter determines who goes next. However, devices closest to the arbiter are implicitly favoured.

Prioritised Daisy-Chaining (Centralised)

To get around the implicit priorities based on distance from the arbiter, many buses have multiple priority levels. For each priority level there is a bus request line and a bus grant line.



It is not technically necessary to wire the level 2 bus grant line serially through devices 1 and 2, since they cannot make requests on it. However, as an implementation convenience, it is easier to wire all the grant lines through all the devices, rather than making special wiring that depends on which device has which priority.

Decentralised Bus Arbitration

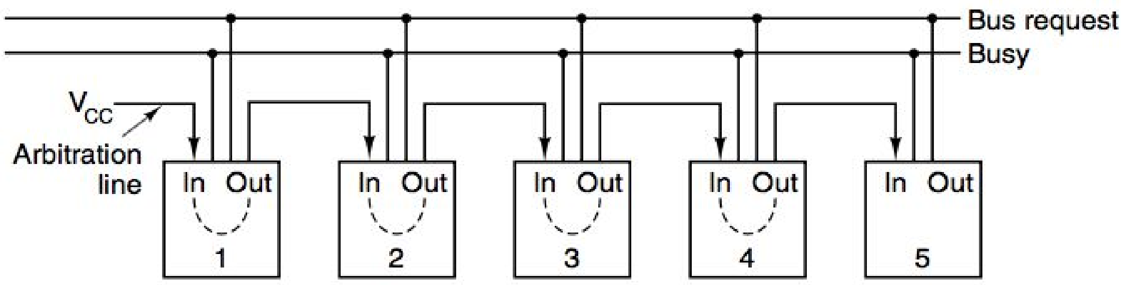


Figure 3-41. Decentralized bus arbitration.

Decentralized bus arbitration uses only three lines, no matter how many devices are present. The first bus line is a wired-OR line for requesting the bus. The second bus line is called **BUSY** and is asserted by the current bus master. The third line is used to arbitrate the bus. It is daisy chained through all the devices. The head of this chain is held asserted by tying it to the power supply.

When no device wants the bus, the asserted arbitration line is propagated through to all devices. To acquire the bus, a device first checks to see if the bus is idle and the arbitration signal it is receiving, **IN**, is asserted.

If **IN** is negated, it may not become bus master, and it negates **OUT**. If **IN** is asserted, however, and the device wants the bus, the device negates **OUT**, which causes its downstream neighbor to see **IN** negated and to negate its **OUT**. Then all downstream devices all see **IN** negated and correspondingly negate **OUT**.

When the dust settles, only one device will have **IN** asserted and **OUT** negated. This device becomes bus master, asserts **BUSY** and **OUT**, and begins its transfer.

Thus, it can be seen that there is somewhat of a 'clock tick', in which devices will send around signals in a decentralised manner until one of them takes control. Only once they have guaranteed control, will they be seen as the master. Data transfer will then commence.

6. Bus Operations

Up until now, we have discussed only ordinary bus cycles, with a master (typically the **CPU**) reading from a slave (typically the memory) or writing to one. In fact, several other kinds of bus cycles exist.

Block Transfers

Normally, one word at a time is transferred. However, when caching is used, it is desirable to fetch an entire cache line (e.g., 8 consecutive 64-bit words) at once. Often block transfers can be made more efficient than successive individual transfers.

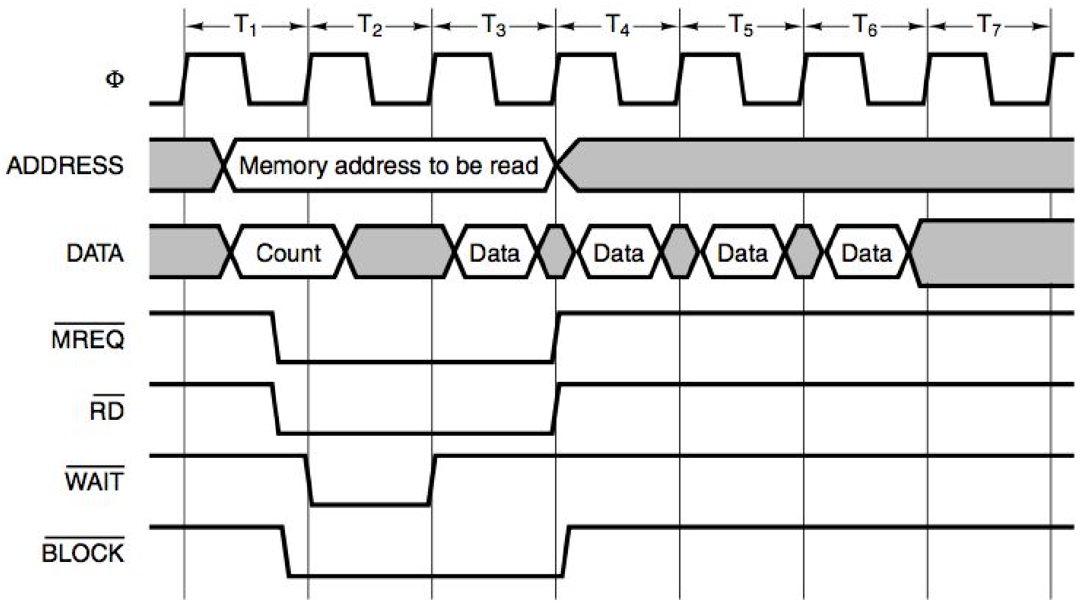


Figure 3-42. A block transfer.

When a block read is started, the bus master tells the slave how many words to expect. Instead of just returning one word, the slave outputs one word during each cycle until the count has been exhausted. In the above example, this allows a block read to take merely 6 cycles instead of 12.

Multiprocessor Bus Lock

A multiprocessor system with two or more **CPU**s on the same bus creates a need to make sure that only one **CPU** at a time uses some critical data structure in memory.
To prevent this situation, multiprocessor systems often have a special read-modify-write bus cycle that allows any **CPU** to read a word from memory, inspect and modify it, and write it back to memory, all without releasing the bus. This type of cycle prevents competing **CPU**s from being able to use the bus and thus interfere with the first **CPU’**s operation.

Interrupt Bus

When the CPU commands an **I/O** device to do something, it usually expects an interrupt when the work is done. The interrupt signaling requires the bus. Since multiple devices may want to cause an interrupt simultaneously, the same kind of arbitration problems are present here that we had with ordinary bus cycles.

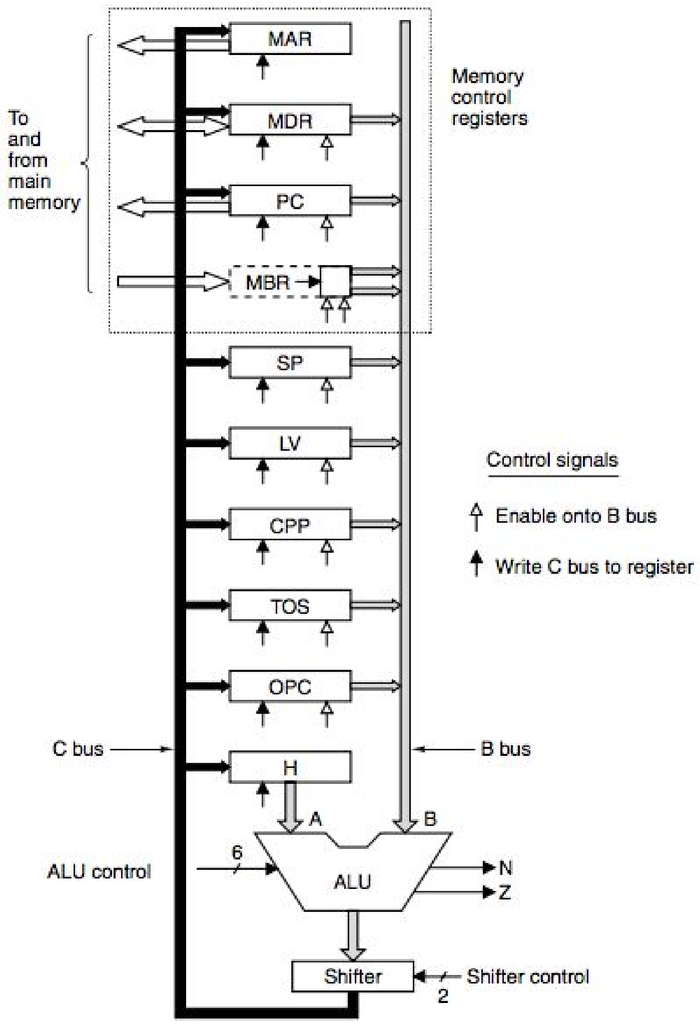
4.1. An Example Microarchitecture

Throughout this chapter, we will be using an example microarchitecture that is fairly similar to that used in most machines. It is, however, simplified for our use.

1. The Data Path

An Example Data Path Diagram

The data path is that part of the **CPU** containing the **ALU**, its inputs, and its outputs. The data path of our example microarchitecture is shown below.



This is fairly similar to the data path used in most machines. It contains a number of 32-bit registers, to which we have assigned symbolic names such as **PC**, **SP**, **MDR**.

Though these names are familiar, these registers are accessible only at the microarchitecture level (by the microprogram), and thus we cannot access them.

They are given these names because they usually hold a value corresponding to the variable of the same name in the instruction set architecture (**ISA**) level architecture.

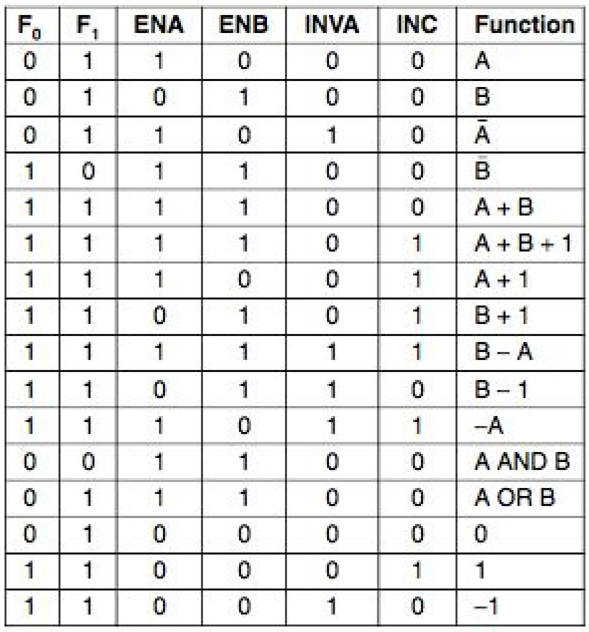
Most registers can drive their contents onto the **B** bus. The output of the **ALU** going into the shifter and then the **C** bus, whose value can be written into one or more registers at the same time for further use. There is no **A** bus for the moment; we will add one later.

Explanation of the Diagram

All the Random Letters in Boxes

:
Each of these are registers. They store data for use later by the ALU.

ALU:
This table below describes the actions of asserting the different flag combinations.



F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

This symbol means that the ALU takes in six control lines, which are as follows:
F0, F1: Determining the ALU Operation
ENA, ENB: Enable Input A, Enable Input B
INVA: Invert Left Input
INC: Forcing Carry into Low-Order Bit

	F0	F1
A AND B	0	0
A OR B	0	1
B COMPLEMENT	1	0
A + B	1	1

It might be easier to look at **B - A** as follows: **-A + B**

-A = ENA, INVA, Carry-One (In 2s Complement, thus negating it)

For Example: A = 001101
INV(A) = 110010 - 1s Compl
Carry-One = 110011 - 2s Compl

This is why we invert A and also send through the (INC) carry flag

B = ENB

Shifter:
The shifter is described above, and is used to shift bits left and right in our input (such as finding a power of two or dividing by two).

The ALU above needs two data inputs:

Input (A):

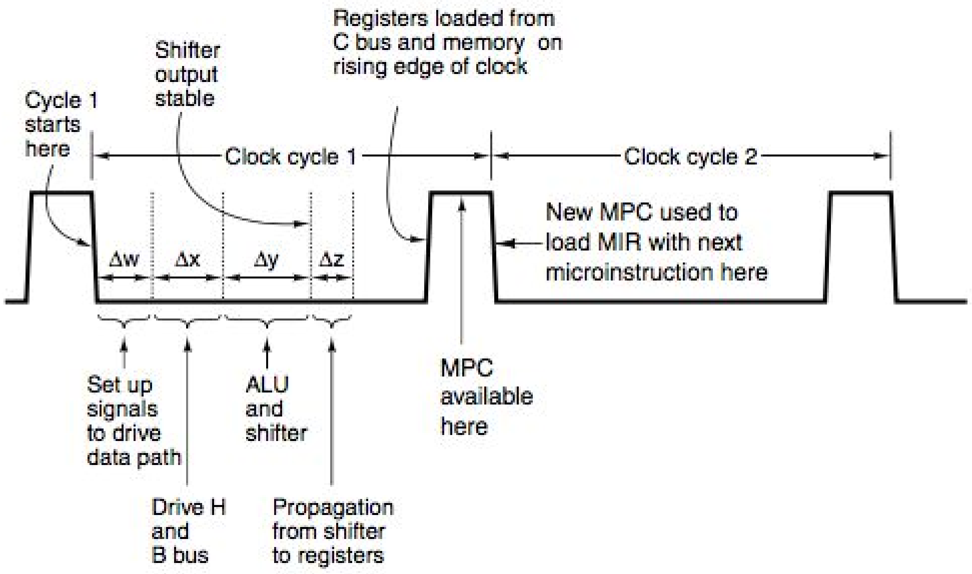
- Attached to the left input is a holding register, H. This input will be assigned by deliberately placing something into register H.
- This is done by choosing an ALU function that only passes the right input (from the B bus) to the ALU output. This is then moved into H by the shifter. An example of this would be adding the ALU inputs, with ENA *unasserted* to make the left zero.

Input (B):

- Attached to the right input is the B bus, which can be loaded from any one of nine sources, indicated by the nine gray arrows touching it.
- This input will be assigned by moving data out of a register and value onto the B bus, as requested.

Data Path Timing

The timing of these events is shown below. Here a short pulse is produced at the start of each clock cycle by the main clock. On the falling edge of the pulse, the bits that will be used by all the gates are set up, and thus the ALU can execute at the asserted (lower) level.


Registers loaded from C bus and memory on rising edge of clock
Shifter output stable
Cycle 1 starts here
Clock cycle 1
Clock cycle 2
New MPC used to load MIR with next microinstruction here
Δw
Δx
Δy
Δz
Set up signals to drive data path
ALU and shifter
MPC available here
Drive H and B bus
Propagation from shifter to registers

Memory Operation

Our machine has two different ways to communicate with memory:

A 32-bit, word-addressable memory port (MDR)
Controlled by two registers, MAR (Memory Address Register) and MDR (Memory Data Register).

An 8-bit, byte-addressable memory port (MBR)
The 8-bit port is controlled by one register, PC (program counter). This reads reads 1 byte per cycle into the low-order 8 bits of MBR like this:

This port can only read data from memory; it cannot write data.

Each of the above is driven by one or two control signals. In our diagram, a write control signal is represented with a ↑ and a read control signal with a ↑.

MAR contains word addresses, so the values 0, 1, 2, etc. refer to consecutive words.
Putting a 2 in MAR and starting a memory read will read out bytes 8–11 (i.e., word 2) from memory and put them in MDR.

PC contains byte addresses, so the values 0, 1, 2, etc. refer to consecutive bytes.
Putting a 2 in PC and starting a memory read will read out byte 2 from memory and put it in the low-order 8 bits of MBR.

Thus, PC and MAR will be used in different cases for accessing different areas in memory.

Using MBR (8-bit):
Data read from memory through the 8-bit memory port are returned in MBR, an 8-bit register. MBR can be gated (i.e., copied) onto the B bus in one of two ways: unsigned or signed.

- Unsigned
  When the unsigned value is needed, the 32-bit word put onto the B bus contains the MBR value in the low-order 8 bits and zeros in the upper 24 bits. Unsigned values are useful for indexing into a table, or when a 16-bit integer has to be assembled from 2 consecutive (unsigned) bytes in the instruction stream.

- Signed
  Converting the 8-bit MBR to a 32-bit word is to treat it as a signed value between −128 and +127 and use this value to generate a 32-bit word with the same numerical value. This conversion is done by duplicating the MBR sign bit (leftmost bit) into the upper 24 bit positions of the B bus, a process known as sign extension. When this option is chosen, the upper 24 bits will either be all 0s or all 1s, depending on whether the leftmost bit of the 8-bit MBR is a 0 or a 1.

2. Microinstructions

The CPU has one single purpose: to fetch assembler instructions from memory, decode and execute them. The "normal program execution flow" is sequential, i.e., after executing one instruction, you execute the following one. This flow determines the data path, and is executed through microinstructions.

A microinstruction is a single instruction in microcode (a very low-level instruction set which is stored permanently in a computer or peripheral controller and controls the operation of the device). It is the most elementary instruction in the computer, such as moving the contents of a register to the ALU.

It takes several microinstructions to carry out one complex machine instruction (CISC).

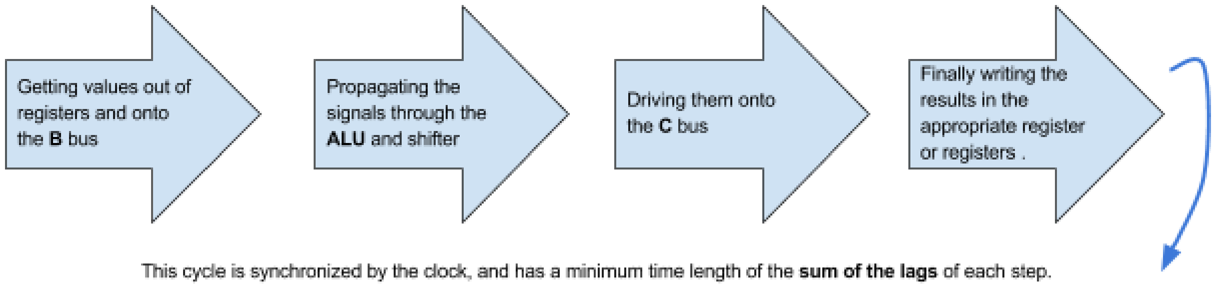
Controlling the Data Path

To control the data path of our example above, we need 29 signals. These can be divided into five types:

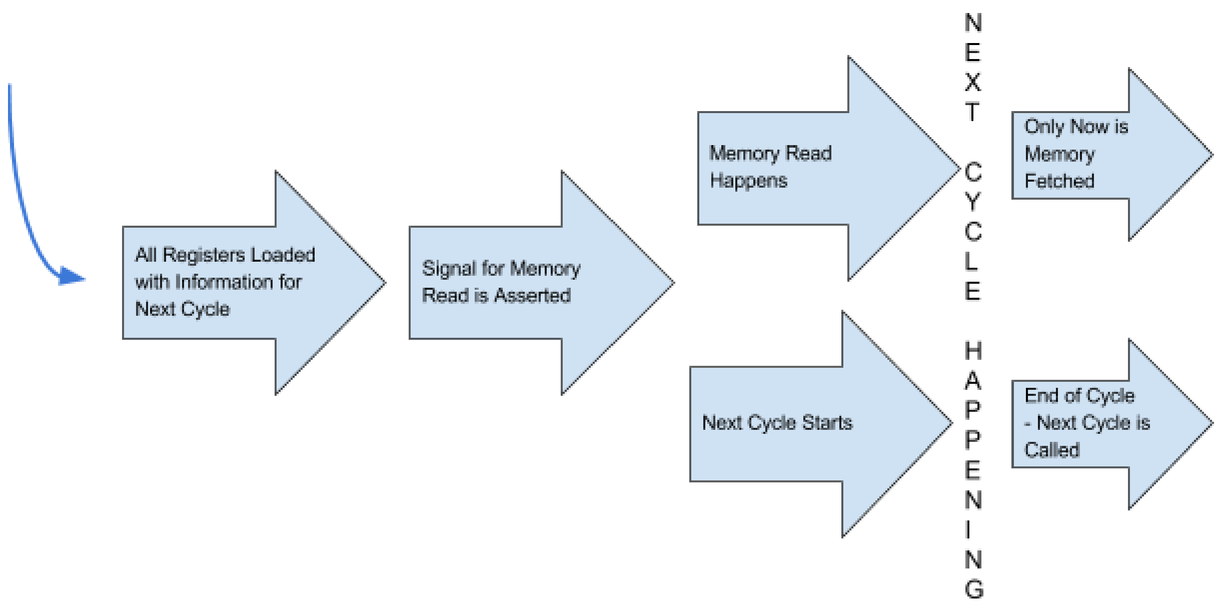
- 9 Signals to control writing data from the **C** bus into registers.
- 9 Signals to control enabling registers onto the **B** bus for **ALU** input.
- 8 Signals to control the **ALU** and shifter functions.
- 2 Signals (not shown) to indicate memory read/write via **MAR** / **MDR** .
- 1 Signal (not shown) to indicate memory fetch via **PC** / **MBR** .

Cycle of Microinstructions

The values of these 29 control signals specify the operations for one cycle of the data path. A cycle consists of:


Getting values out of registers and onto the B bus
Propagating the signals through the ALU and shifter
Driving them onto the C bus
Finally writing the results in the appropriate register or registers .
This cycle is synchronized by the clock, and has a minimum time length of the sum of the lags of each step.

In addition, if a memory read data signal is asserted, the memory operation is started at the end of the data path cycle, after **MAR** has been loaded. The memory data are available at the very end of the following cycle in **MBR** or **MDR** and can be used in the cycle after that.

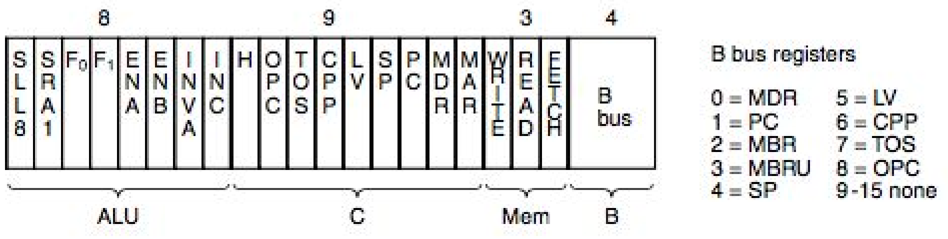

All Registers Loaded with Information for Next Cycle
Signal for Memory Read is Asserted
Memory Read Happens
Next Cycle Starts
NEXT CYCLE HAPPENING
Only Now is Memory Fetched
End of Cycle - Next Cycle is Called

In other words, a memory read initiated at the end of cycle *k* delivers data that cannot be used in cycle *k + 1*, but only in cycle *k + 2* or later.

This means that, as registers are read on a low-cycle, that old data (in MBR and MDR) is still accessible <u>until</u> the data is overwritten during a clock transition period (rising edge).

Remember, we are talking best case scenario. The *assumption* that the memory takes one cycle to operate is equivalent to assuming that the **level 1** cache hit rate is 100%. This assumption is ***never*** true, ***but*** the complexity introduced by a variable-length memory cycle time is too complicated for our little squirrel brains.

In essence, our current control looks something like the below.

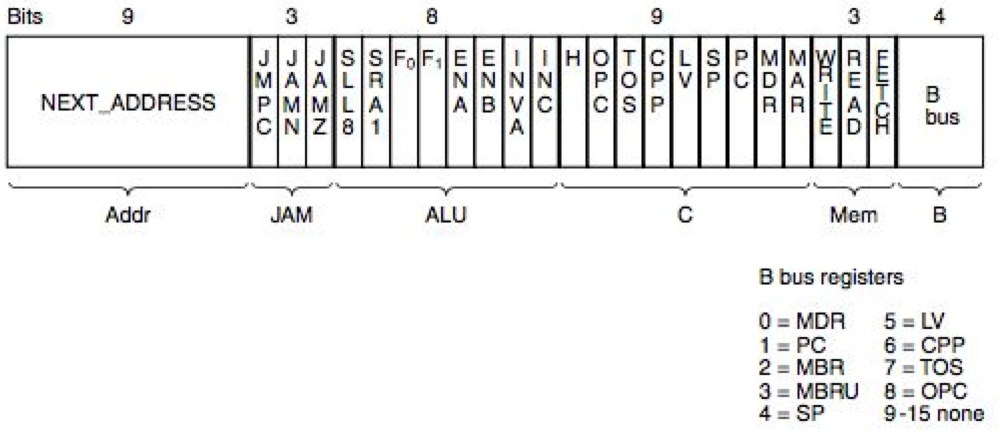


If you're a maths person, you would have done some calculations and seen that we've only used 24 signals thus far? 9 + 4 + 8 + 2 + 1 = 24 signals. What about the other 5?

Well, in fact, **these are our 29 signals**, with B being compressed from 9 signals into 4. There are only nine possible input registers that can drive the B bus and we can therefore encode the B bus information in 4 bits and use a decoder to generate the 16 control signals, 7 of which will not be needed.

Describing the Next Cycle

These 24 bits **only** control the data path for *one cycle*. The second part of the control is to determine what is to be done on the following cycle. To include this in the design of the controller, we will create a new format for describing the operations to be performed using the 24 control bits plus two additional fields: The **NEXT_ADDRESS** field and the **JAM** field.



The figure above is a possible format, divided into the six groups, and contains the following 36 signals:

- **Addr** – Contains the address of a potential next microinstruction.
- **JAM** – Determines how the next microinstruction is selected.
- **ALU** – **ALU** and shifter functions.
- **C** – Selects which registers are written from the **C** bus.
- **Mem** – Memory functions.
- **B** – Selects the **B** bus source; it is encoded as shown. There are only nine possible input registers that can drive the B bus. Therefore, we can encode the B bus information in 4 bits and use a decoder to generate the 16 control signals, 7 of which are not needed.

Ordering of Instruction Groups

The ordering of the groups has been carefully chosen to minimize line crossings on our circuit board. Line crossings in schematic diagrams like the one below often correspond to wire crossings on chips, which cause trouble in two-dimensional designs and are best minimized.

3. Microinstruction Control: The Mic-1

Quick Recap

Before we can discuss the actual micro-program itself, we need to understand how the CPU fetch the next microinstruction. Once again, flow control is the flow of program execution. The "normal program execution flow" is sequential, i.e., after executing one instruction, you execute the following one.

To understand how the CPU run/execute the micro-program, we need to understand how the micro program flow control. First: where is the micro program?

Recall that a micro-program is a sequence of microinstructions that controls the behaviour of the datapath (and thus also the CPU). Recall also that the CPU has one single purpose: to fetch assembler instructions from memory, decode and execute them.

Therefore, the micro-program is very specific. Normally, programs are stored in the memory. But if you think about it, it would not be practical to store the microprogram in memory because it would be too slow to fetch micro-instructions each time a CPU cycle is requested. Fortunately, the micro program is very small (because all it does is to instruct the CPU to perform the steps in the assembler instruction execution cycle, which only consists of a few steps).

Thus, the microprogram is stored inside the CPU on something called the Control Store. This is detailed below.

Determining What Happens When

It's now important to describe how we decide the signals set per cycle.

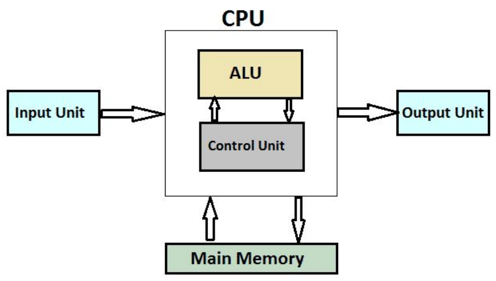
These decisions are made by something called the **sequencer**. It steps through the sequence of operations necessary to execute a single **instruction set architecture** instruction. The sequencer must produce two kinds of information each cycle:

1. The state of every control signal in the system.
2. The address of the microinstruction that is to be executed next.

Below is a detailed block diagram of the complete microarchitecture of our example machine, which we will call the **Mic-1**. It looks scary as shit, but it's actually not too bad.

Description of the Block Diagram of Mic-1

When you fully understand every box and every line in the block diagram figure below (after this simple one), you will be well on your way to understanding the microarchitecture level.



From school I.T. or even from general knowledge, you will know that a CPU consists of, in veeeery basic terms, the **ALU and Control Unit**. The ALU makes up the **Data Path**, whilst the Control Unit makes up the **Control Section**.

<u>The block diagram below consists of these two sections in detail.</u>

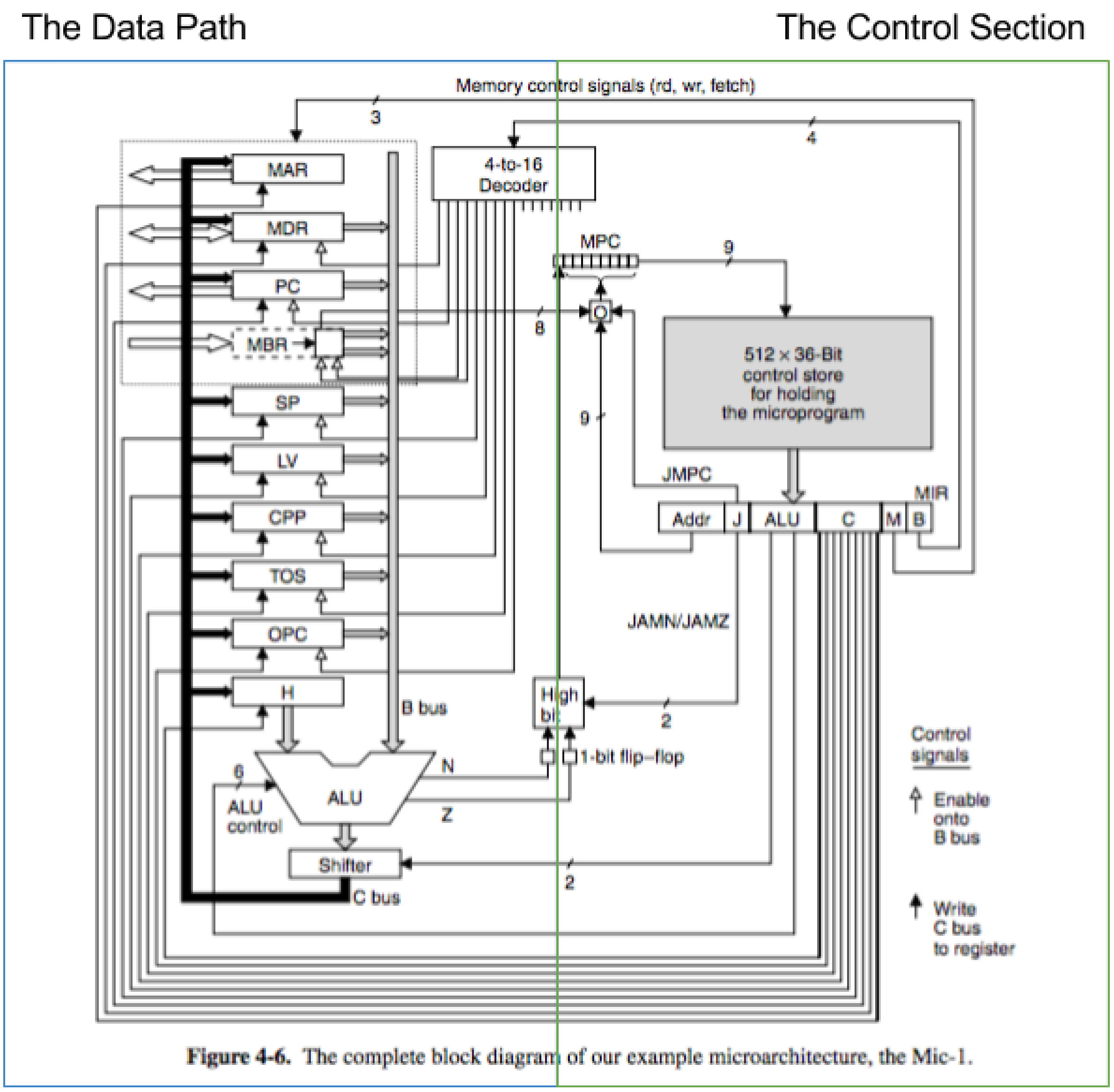


Figure 4-6. The complete block diagram of our example microarchitecture, the Mic-1.

The Data Path (Left)
This is our 'ALU'. We now know that it consists of more than merely the Arithmetic Logic Unit, however it's usually grouped together.

The data path is the functional units (such as arithmetic logic units or multipliers, that perform data processing operations), registers, and buses. This is what we've been studying above. Everything from chapter 4.1.1 describes this process, so I won't re-summarise it.

The Control Section (Right)
The control section does just that, control what happens with the data path and in the registers. It comprises of three main paths, all of which are necessary in correctly assigning jobs to the components of the Data Path.

➔ **The Control Store:** The microprogram is stored in a ROM. This ROM is the largest and most important item, and is called the control store. It is convenient to think of it as a memory that holds the complete microprogram (a microinstruction program that controls the functions of a central processing unit or peripheral controller of a computer).

  In general, we will refer to it as the control store, to avoid confusion with the main memory, accessed through MBR and MDR. Functionally, however, the control store is a memory that simply holds microinstructions instead of instruction set architecture instructions.

  Each microinstruction explicitly specifies its successor and since the control store is functionally a (read-only) memory, it only needs its own memory address register and memory data register.

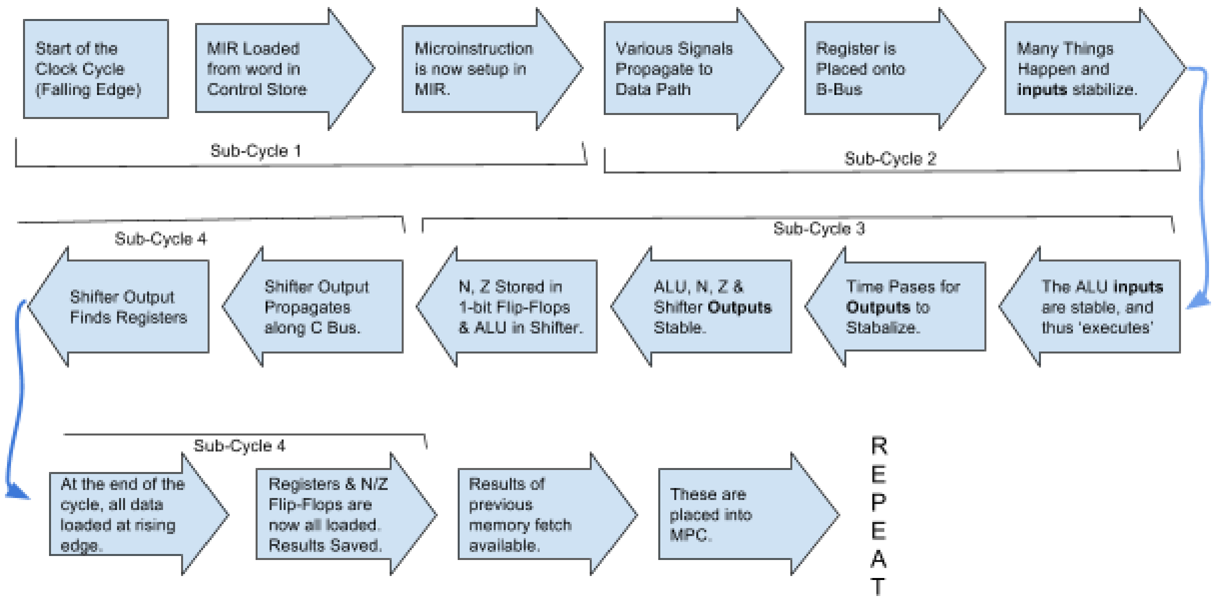
➔ **MPC (MicroProgram Counter)** - Points to the Next Instruction
  The MPC is used to determine which instruction to execute next. It is built up using a combination of input from the *MBR on the Data Path*, as well as from the previous microinstruction itself (from the ADDR and JAM registers).

➔ **MIR (MicroInstruction Register)** - Holds Current Instruction
  This is in the same format as before. The Addr and J (for JAM) groups control the selection of the next microinstruction. The ALU group contains the 8 bits that select the ALU function and drive the shifter. The C bits cause individual registers to load the ALU output from the C bus. The M bits control memory operations. Finally, the last 4 bits drive the decoder that determines what goes onto the B bus. In this case we have chosen to use a standard 4-to-16 decoder, even though only nine possibilities are required

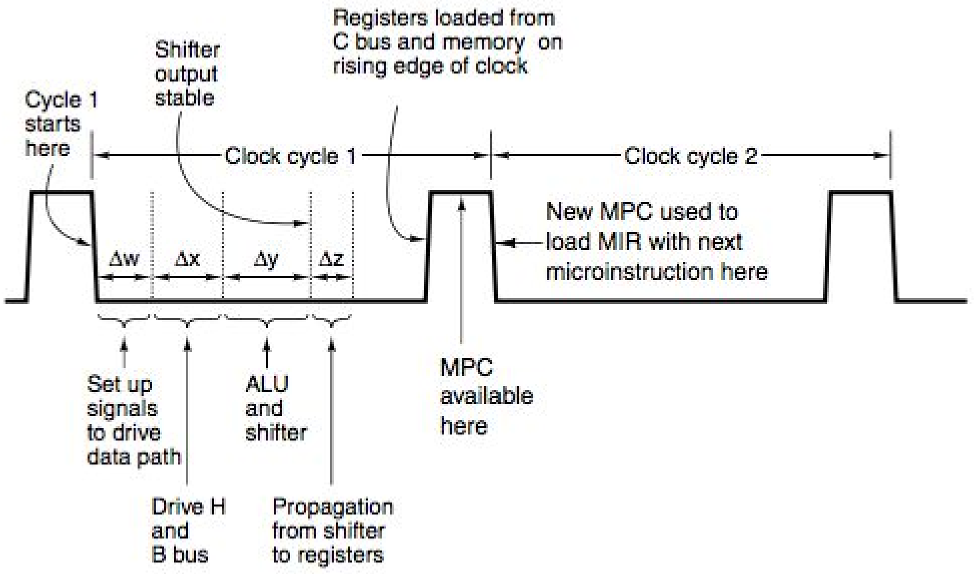
Operation of the Block Diagram of Mic-1 for Driving the Data Path

One of two operations involves driving the data path for the CPU execution. This cycle is detailed below in this really badly-drawn flowchart.

Below, **stable** means that the inputs are constant and ready to use.


Start of the Clock Cycle (Falling Edge)
MIR Loaded from word in Control Store
Microinstruction is now setup in MIR.
Various Signals Propagate to Data Path
Register is Placed onto B-Bus
Many Things Happen and inputs stabilize.
Sub-Cycle 1
Sub-Cycle 2
Sub-Cycle 4
Sub-Cycle 3
Shifter Output Finds Registers
Shifter Output Propagates along C Bus.
N, Z Stored in 1-bit Flip-Flops & ALU in Shifter.
ALU, N, Z & Shifter Outputs Stable.
Time Pases for Outputs to Stabalize.
The ALU inputs are stable, and thus 'executes'
Sub-Cycle 4
At the end of the cycle, all data loaded at rising edge.
Registers & N/Z Flip-Flops are now all loaded. Results Saved.
Results of previous memory fetch available.
These are placed into MPC.
REPEAT

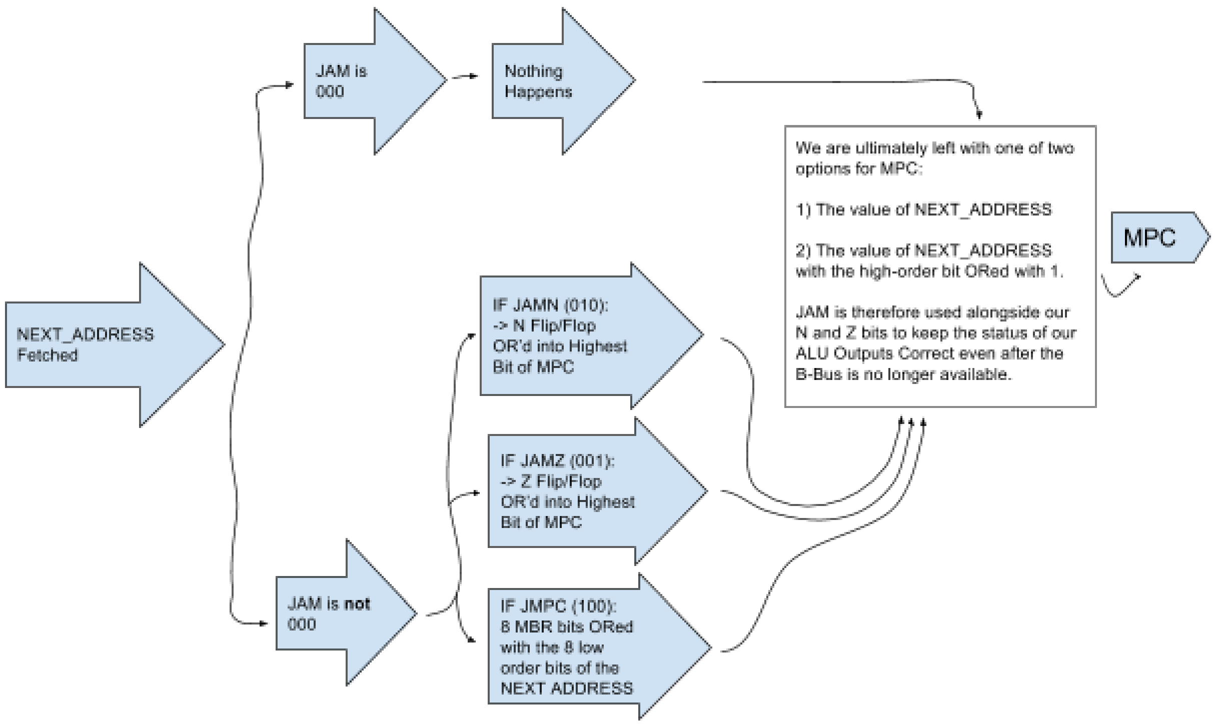
The Sub-Cycles above are all part of the clock cycle as described below:


Registers loaded from C bus and memory on rising edge of clock
Shifter output stable
Cycle 1 starts here
Clock cycle 1
Clock cycle 2
Δw
Δx
Δy
Δz
New MPC used to load MIR with next microinstruction here
Set up signals to drive data path
ALU and shifter
MPC available here
Drive H and B bus
Propagation from shifter to registers

Operation of the Block Diagram of Mic-1 for Determining Next Instruction

In parallel with driving the data path, the microprogram also has to determine which microinstruction to execute next, as they need not be executed in the order they happen to appear in the control store.

This begins as soon as MIR has been loaded, and is stable.


NEXT_ADDRESS Fetched
JAM is 000
Nothing Happens
JAM is not 000
IF JAMN (010): -> N Flip/Flop OR'd into Highest Bit of MPC
IF JAMZ (001): -> Z Flip/Flop OR'd into Highest Bit of MPC
IF JMPC (100): 8 MBR bits ORed with the 8 low order bits of the NEXT ADDRESS
We are ultimately left with one of two options for MPC:
1) The value of NEXT_ADDRESS
2) The value of NEXT_ADDRESS with the high-order bit ORed with 1.
JAM is therefore used alongside our N and Z bits to keep the status of our ALU Outputs Correct even after the B-Bus is no longer available.
MPC

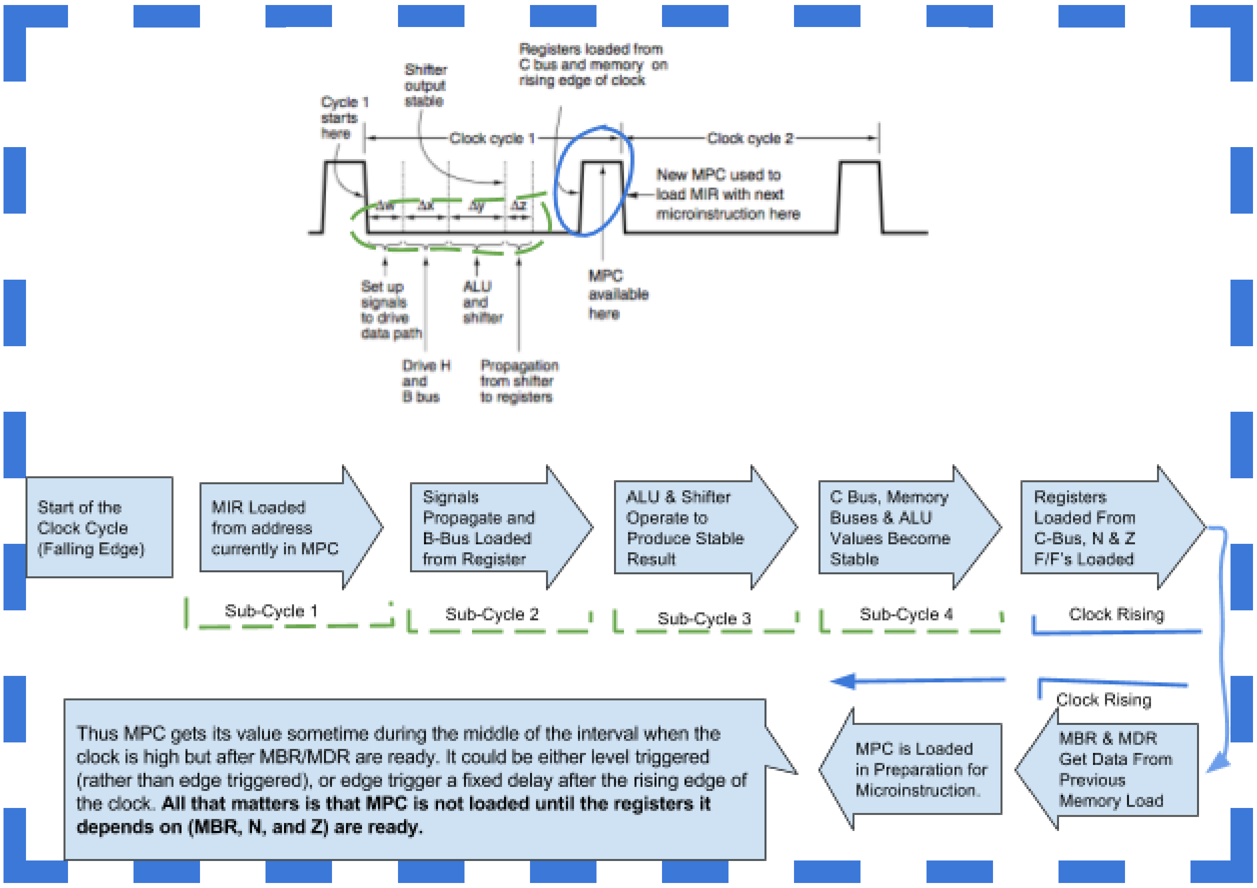
Whilst JAMN and JAMZ is pretty self explanatory (from the diagram), JMPC is rather confusing.

If it is set, the 8 MBR bits are bitwise ORed with the 8 low-order bits of the NEXT ADDRESS field coming from the current microinstruction. The result is sent to MPC. This is important!

The box with the label “O” in our figure does an OR of MBR with NEXT_ADDRESS if JMPC is 1, but just passes NEXT_ADDRESS through to MPC if JMPC is 0.

In a typical use, MBR contains an opcode, so the use of JMPC will result in a unique selection for the next microinstruction to be executed for every possible opcode. **<u>Thus, this is what allows the data path to change the next instruction, based on data fetched.</u>**

This is explained in detail below. This is crucial, and understanding the clock cycle and sub-cycles equally so.


Registers loaded from C bus and memory on rising edge of clock
Shifter output stable
Cycle 1 starts here
Clock cycle 1
Clock cycle 2
New MPC used to load MIR with next microinstruction here
Δw
Δx
Δy
Δz
Set up signals to drive data path
ALU and shifter
MPC available here
Drive H and B bus
Propagation from shifter to registers
Start of the Clock Cycle (Falling Edge)
MIR Loaded from address currently in MPC
Signals Propagate and B-Bus Loaded from Register
ALU & Shifter Operate to Produce Stable Result
C Bus, Memory Buses & ALU Values Become Stable
Registers Loaded From C-Bus, N & Z F/F's Loaded
Sub-Cycle 1
Sub-Cycle 2
Sub-Cycle 3
Sub-Cycle 4
Clock Rising
Clock Rising
MBR & MDR Get Data From Previous Memory Load
MPC is Loaded in Preparation for Microinstruction.

Thus MPC gets its value sometime during the middle of the interval when the clock is high but after MBR/MDR are ready. It could be either level triggered (rather than edge triggered), or edge trigger a fixed delay after the rising edge of the clock. **All that matters is that MPC is not loaded until the registers it depends on (MBR, N, and Z) are ready.**

The above shows the way that our NEXT_ADDRESS is loaded into the MPC.

4.2. An Example ISA: IJVM

What is an ISA? An ISA is an instruction set architecture, which is an abstract model of a computer. It is also referred to as architecture or computer architecture. We kinda program on this (and as you recall from the Jasmin->JVM program, this is the type of stuff we wrote on).

Thus, an ISA is the first computing level for which we are able to program. These, regardless of hardware, have their own instruction sets - which are then **interpreted** and **executed** by the microprogram running on the microarchitecture.

In terms of our example, *IJVM is an instruction set architecture created by Andrew Tanenbaum for his MIC-1 architecture. It is used to teach assembly basics in his book Structured Computer Organization.*

IJVM is mostly a subset of the JVM assembly language that is used in the Java platform. This instruction set is so simple that it's difficult to write complex programs in it (for example, no shift instructions are provided).

We mostly used this for our compiler project.

1. Stacks

What is a Stack?

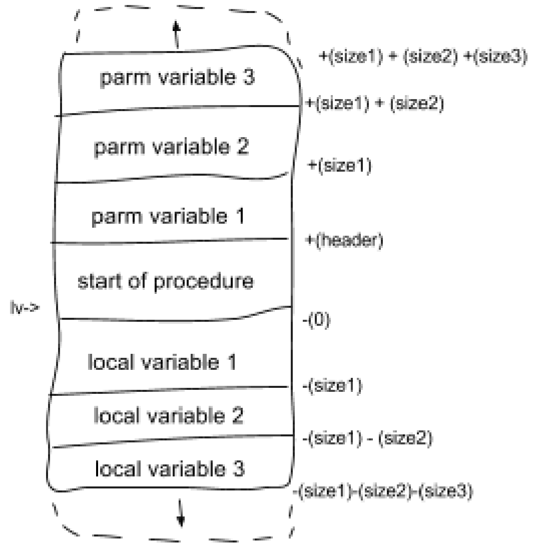
You should already know (unless you've managed this term without using one, which is more impressive than shocking), that a stack is a LIFO (last in, first out) object onto which objects or data can be 'Pushed' and 'Popped'.

Why is a stack useful?

Function Stack

Virtually all programming languages support the concept of procedures (methods), which have local variables. In terms of normal scope (determining where certain variables can be accessed), these variables can be accessed from inside the method but stop being accessible once the method has been returned.

However, where do we store these variables in memory? If we were to give each variable an **absolute** memory address, we'd run into a problem if we were to call the same function recursively, i.e. calling itself, the registers would then be competed over by the different functions (and that's bad and stuff for reasons because then everything fucks out).



Instead, an area of memory, called the **stack**, is used for variables. Individual variables don't get absolute addresses in it, but rather are given relative addresses based on the start of the procedure from which it is called.

I.e: if you look to the left, you can see that variables given as parameters are addressed as the start of the procedure plus (+) the size of the respective variables

Local variables are addressed as the start of the procedure minus (-) the size of the respective variables.

A pointer, LV, constantly points to the start of the current procedure. Should another procedure be called, its variables will begin directly on top of the variables of the previous function, etc. The LV will then point to the new function. As soon as the function returns, the pointer will, once again, point to the function calling it.

This is demonstrated below, where a,b,c,d refer to different function variables.

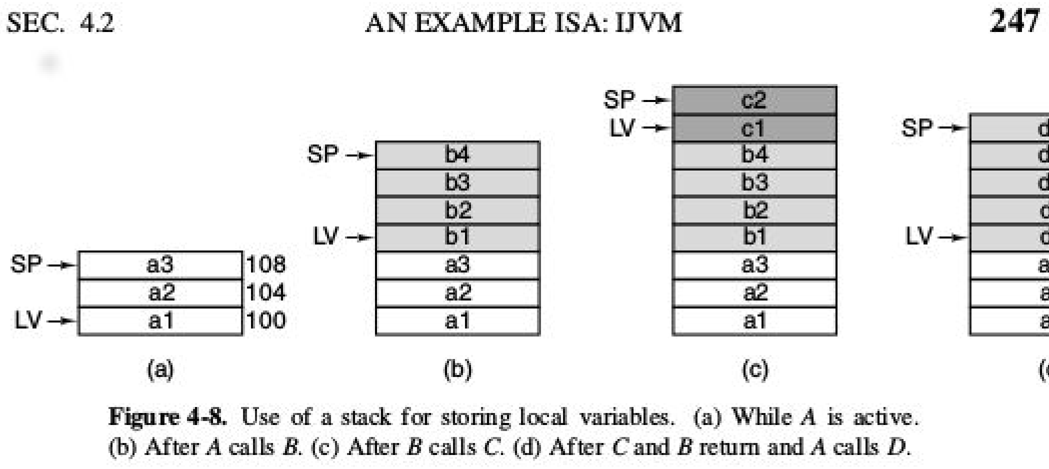


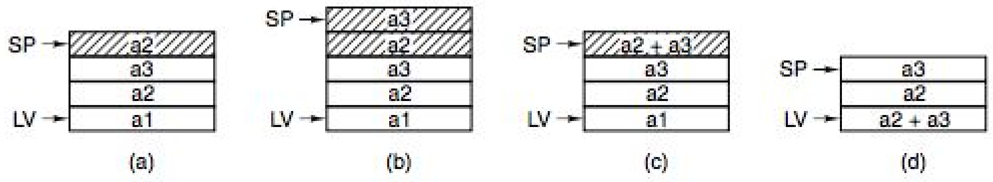
Figure 4-8. Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

Operand Stack

Besides holding local variables, stacks have another use. They can hold operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the operand stack. Suppose, for example, that before calling B, A has to do the computation

a1 = a2 + a3

In order to do this operation, a2 **(a)** and a3 **(b)** will be pushed onto the stack, and then a SUM instruction is called to pop a2 and a3, calculate, and then push the result a1 **(c)**. This is now stored on top of the stack. This is now popped off the stack and stored back in a local variable **(d)**.



2. The IJVM Memory Model

We can view memory in either of two ways: an array of 4,294,967,296 bytes (4 GB) or an array of 1,073,741,824 words, each consisting of 4 bytes.

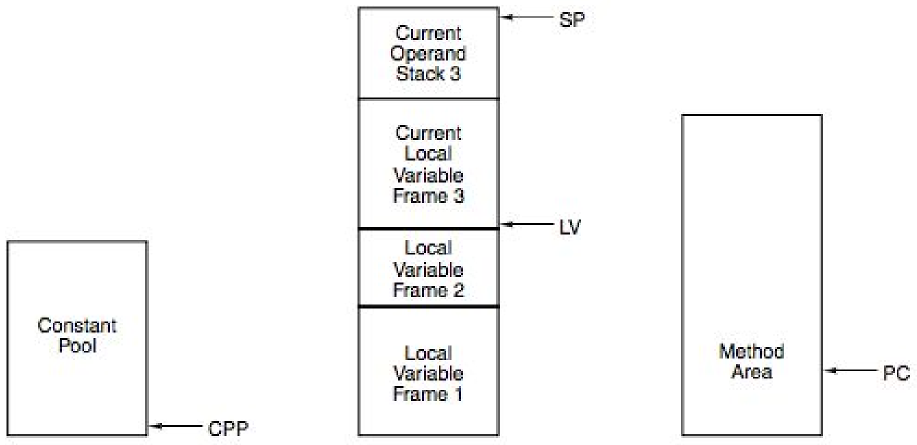
Within this memory, at any time, the following areas of memory are defined:

1. The Constant Pool. This area cannot be written by an IJVM program and consists of constants, strings, and pointers to other areas of memory that can be referenced. It is loaded when the program is brought into memory and not changed afterward. There is an implicit register, CPP, that contains the address of the first word of the constant pool.

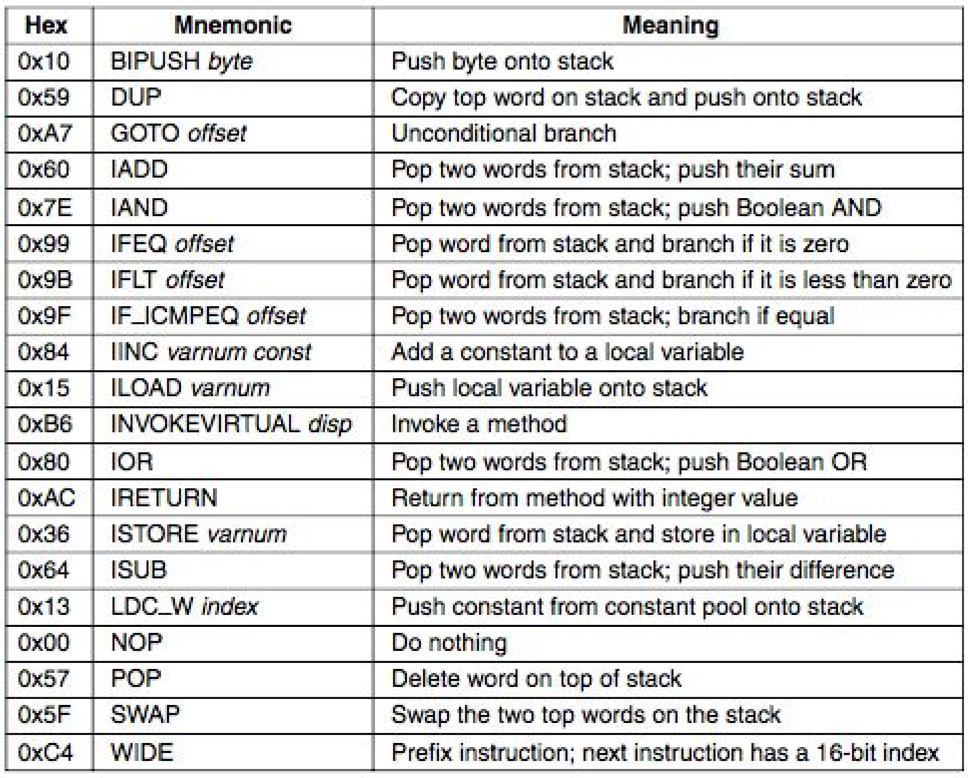
2. The Local Variable Frame. For each call of a method, an area is allocated for storing variables during the lifetime of the method's runtime. It is called the local variable frame. At the beginning of this frame reside the parameters (also called arguments) with which the method was invoked. The local variable frame does not include the operand stack, which is separate. Our implementation chooses to implement the operand stack immediately above the local variable frame. An implicit register contains the address of the first location in the local variable frame. We will call this register ***LV***. The parameters passed at the invocation of the method are stored at the beginning of the local variable frame.

3. The Operand Stack. The stack frame is guaranteed not to exceed a certain size, computed in advance by the Java compiler. The operand stack space is allocated directly above the local variable frame, as illustrated in our stack above. In our implementation, it is convenient to think of the operand stack as part of the local variable frame. In any case, an implicit register contains the address of the top word of the stack. Notice that, unlike CPP and LV, this pointer, SP, changes during the execution of the method as operands are pushed onto the stack or popped from it.

4. The Method Area. Finally, there is a region of memory containing the program, referred to as the ''text'' area in a UNIX process. An implicit register contains the address of the instruction to be fetched next. This pointer is referred to as the Program Counter, or PC. Unlike the other regions of memory, the method area is treated as a byte array.



3. The IJVM Instruction Set



Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

An instruction set is a definition of what certain instructions do when called. It also specifies what each instruction requires as part of the call. Each instruction consists of an opcode and sometimes an operand, such as a memory offset or a constant. The first column above gives the hexadecimal encoding of the instruction. The second gives its assembly-language mnemonic. The third gives a brief description of its effect. See Page 265 in the textbook for detailed explanations. We did this in our project, so I don't think it's necessary to go in depth.

The operands byte, const, and varnum are 1 byte. The operands disp, index, and offset are 2 bytes.

Types of Instructions

Push Instructions

These sources include the constant pool (**LDC W**), the local variable frame (**ILOAD**), and the instruction itself (**BIPUSH**). A variable can also be popped from the stack and stored into the local variable frame (**ISTORE**).

Arithmetic & Logic Instructions

Two arithmetic operations (**IADD** and **ISUB**) as well as two logical (Boolean) operations (**IAND** and **IOR**) can be per- formed using the two top words on the stack as operands. In all the arithmetic and logical operations, two words are popped from the stack and the result pushed back onto it.

Branch Instructions

Four branch instructions are provided, one unconditional (**GOTO**) and three conditional ones (**IFEQ**, **IFLT**, and **IF ICMPEQ**). All the branch instructions, if taken, adjust the value of **PC** by the size of their (16-bit signed) offset, which follows the opcode in the instruction.

Other Instructions

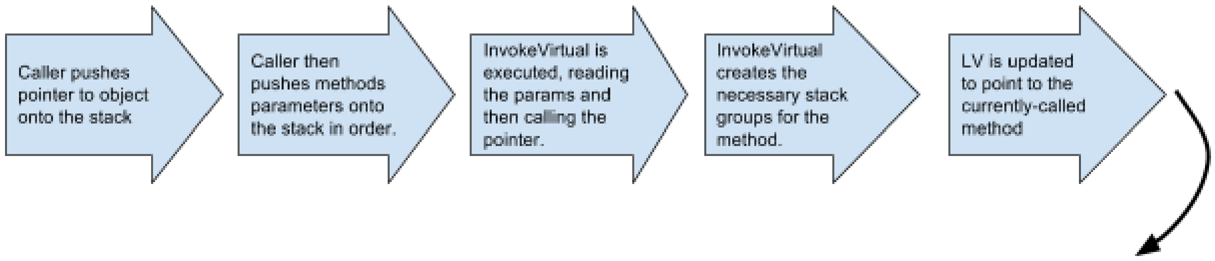
There are also ***IJVM*** instructions for swapping the top two words on the stack (**SWAP**), duplicating the top word (**DUP**), and removing it (**POP**).

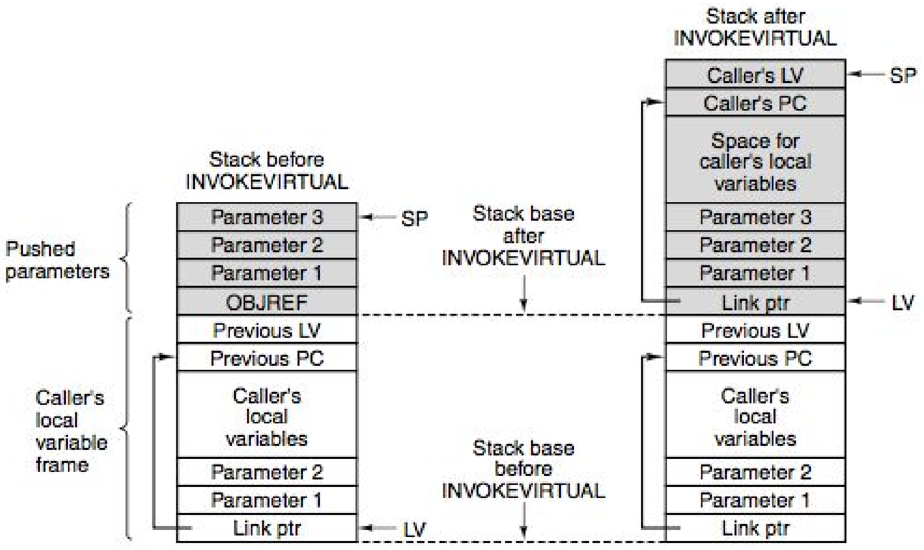
Invocation Instructions

Finally, there is an instruction (**INVOKEVIRTUAL**) for invoking another method, and another instruction (**IRETURN**) for exiting the method and returning control to the method that invoked it.

Due to the complexity of the mechanism, the definition has been simplified to make it possible to produce a straightforward mechanism for invoking a call and return. For this, don't think object oriented programming, because by simplifying, we remove this.

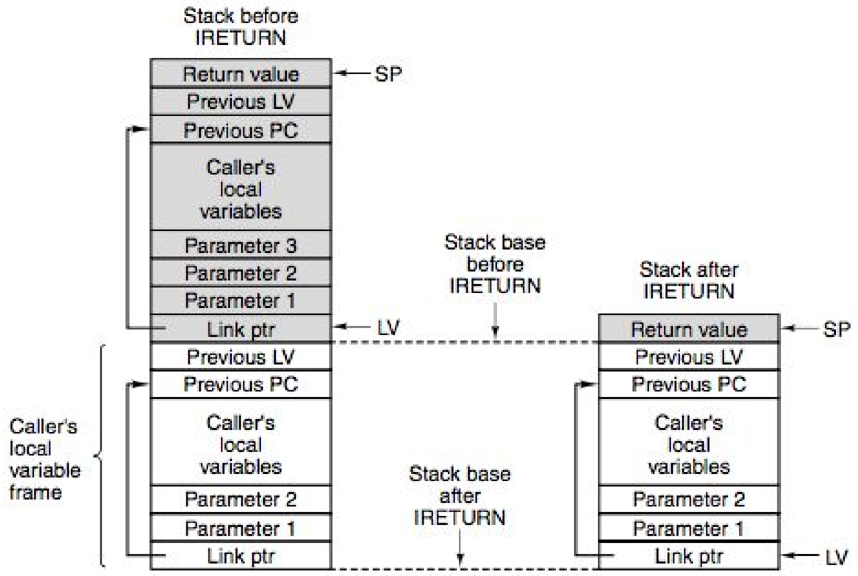
Invoking a Method


Caller pushes pointer to object onto the stack
Caller then pushes methods parameters onto the stack in order.
InvokeVirtual is executed, reading the params and then calling the pointer.
InvokeVirtual creates the necessary stack groups for the method.
LV is updated to point to the currently-called method



Returning a Method

The **IRETURN** instruction reverses the operations of the **INVOKEVIRTUAL** in- struction, as shown below. It deallocates the space that was used by the returning method. It also returns the stack to the former state, except for the fact that the returned value is placed on top of the stack.



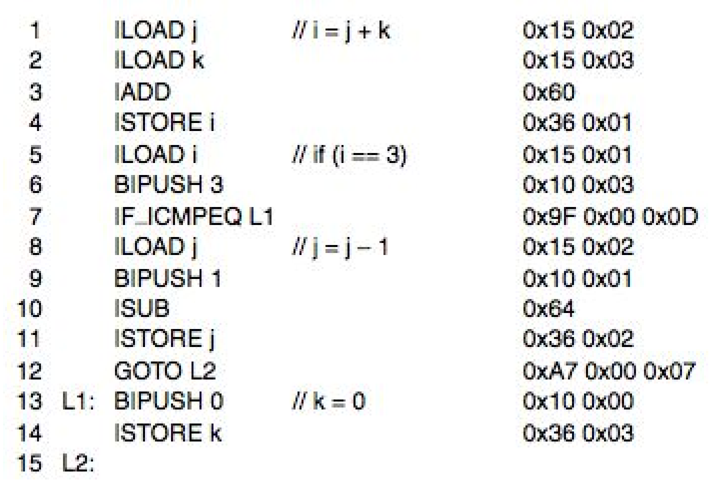
4. Compiling Java to IJVM

Java is a language written to be executed on the Java Virtual Machine (JVM), which is inherently based on IJVM.

Example Program

```
i = j + k;
if (i == 3)
      k = 0;
else
      j = j - 1;
```

The example program above can be written in IJVM byte-code as follows:



```
     ILOAD j        // i = j + k      0x15 0x02
     ILOAD k                          0x15 0x03
     IADD                             0x60
     ISTORE i                         0x36 0x01
     ILOAD i        // if (i == 3)    0x15 0x01
     BIPUSH 3                         0x10 0x03
     IF_ICMPEQ L1                     0x9F 0x00 0x0D
     ILOAD j        // j = j – 1      0x15 0x02
     BIPUSH 1                         0x10 0x01
     ISUB                             0x64
     ISTORE j                         0x36 0x02
     GOTO L2                          0xA7 0x00 0x07
L1:  BIPUSH 0       // k = 0          0x10 0x00
     ISTORE k                         0x36 0x03
L2:
```

The compiled code is relatively straightforward after doing the project. First j and k are pushed onto the stack, and the result is summed into i. Then i and the constant 3 is pushed onto the stack, and compared. If they are equal, we jump to L1 where k is set to 0. If they are unequal, the compare fails and the code following is executed. When it is done, it jumps to L2 where the then and else parts merge.

4.3. An Example Implementation

You may ask: Right, so what does a program running on the **micro**architecture (machine level, the MIC-1,2,3,4 implementations) and interpreting the **macro**architecture (instruction set etc) look like, and how does it work?

Alright, probably not, but you're about to find out anyways.

Before we can answer these questions, we must carefully consider the notation we will use to describe the implementation. Once again, this is a very long chapter and should be read thoroughly in the textbook (summarising this is kinda like trying to pick a fish up from out of water; as soon as you think you have it; it slaps you in the face and you have to start all over again). Bad analogy, I know. It's late.

1. Microinstructions and Notation

In principle, we could describe the control store in binary, 36 bits per word. But that's an effort. As in normal programming languages, it's better to introduce notation that simplifies life - ignoring irrelevant details that would be determined automatically irregardless.

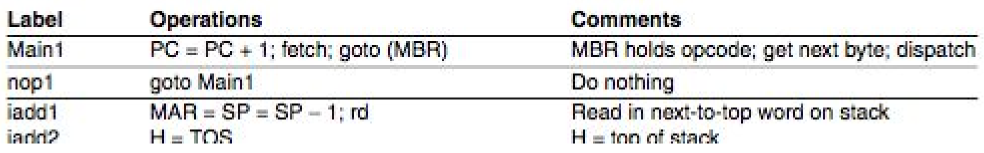
One aspect where this issue is important is the choice of addresses. Since the memory is not logically ordered, there is no natural ''next instruction'' as we specify a sequence of operations. Much of the power of this control organization derives from the ability of the assembler (turning from Macro- into Micro-architecture) to select addresses efficiently.

We therefore begin by introducing a simple symbolic language that fully describes each operation, however simply ignore how all addresses may have been determined. **Our notation specifies all the activities that occur in a single clock cycle in a single line.**

This is described in-depth in the textbook, and would be impossible to repeat here. Basically, each instruction is encoded as a micro-operation list. Thus, when an instruction is called, it's operation list is executed.

For example, to copy something from SP to MDR, we can do the following:
$MDR = H + SP$ (*where H is our H register and SP is a register pushed to the B Bus*).

This type of thing is repeated for every instruction, and is summarised in the table in the textbook. An extract can be found below:



Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack

2. Implementation of IJVM using Mic-1

What does it mean to Implement?

We have finally reached the point where we can put all the pieces together. Figure 4-17 in the textbook is the microprogram that runs on Mic-1 and interprets IJVM. It is a surprisingly short program—only 112 microinstructions total.

Given for each microinstruction are: the symbolic label, the actual microcode, and a comment. Note that consecutive microinstructions are not necessarily located in consecutive addresses in the control store, as we have already pointed out.

Register Definitions:

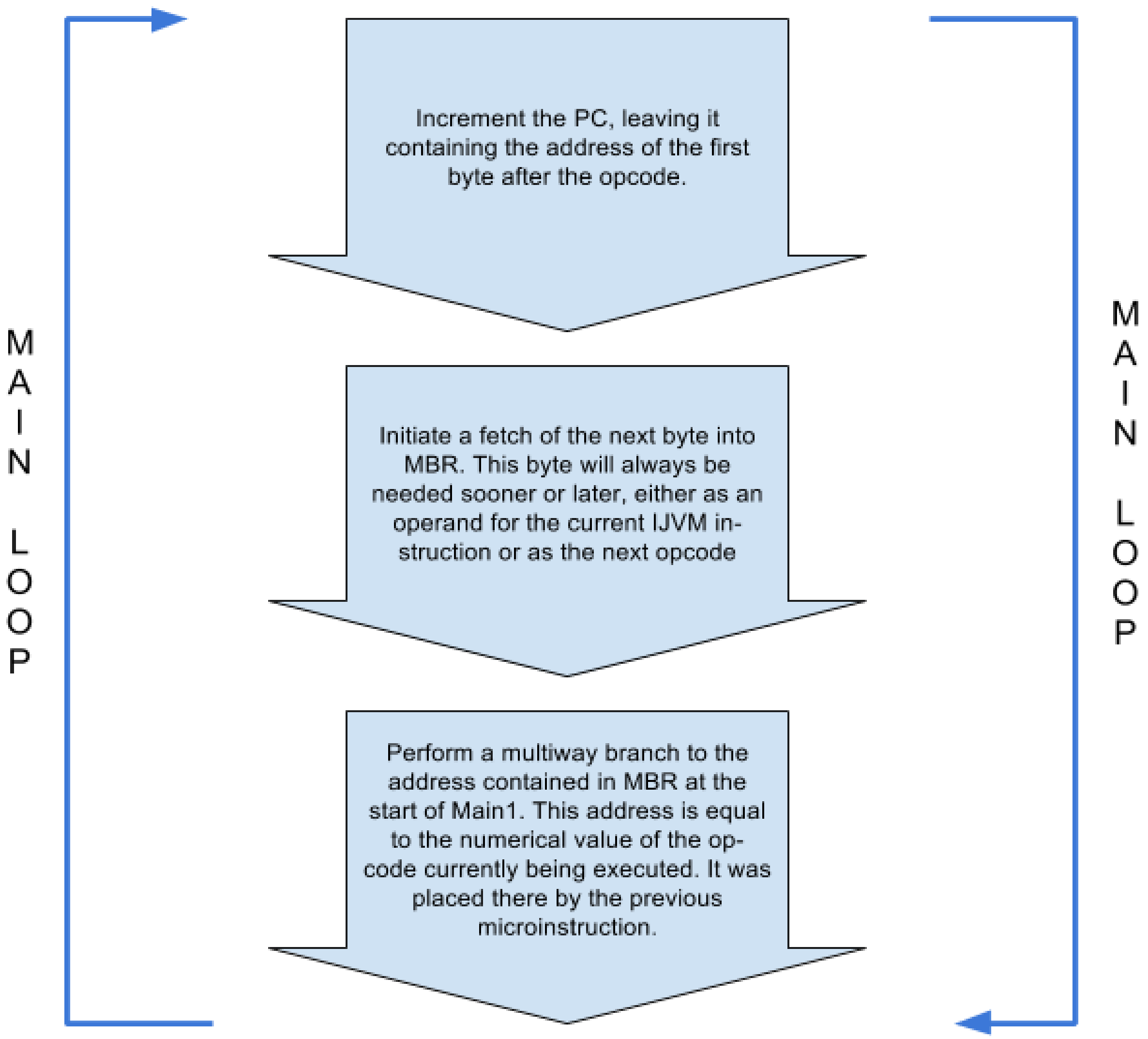
Each register usually has a purpose. These are summarised below.

CPP	Pointer to Constant Pool
LV	Pointer to Local Variables
SP	Pointer to the Top of Stack
PC	Holds Address of Next Byte to be Fetched
MBR	Sequentially holds the bytes of the instruction stream as they come in from memory to be interpreted.
TOS	Extra Register, see below.
OPC	Extra Register, see below.

At the beginning and end of each instruction, TOS contains the value of the memory location pointed to by SP, the top word on the stack. This value is redundant since it can always be read from memory, but having it in a register often saves a memory reference. For a few instructions maintaining TOS means more memory operations. For example, the POP instruction throws away the top word and therefore must fetch the new top-of-stack word from the memory into TOS.

The OPC register is a temporary (i.e., scratch) register. It has no preassigned use. It is used, for example, to save the address of the opcode for a branch instruc- tion while PC is incremented to access parameters. It is also used as a temporary register in the IJVM conditional branch instructions.

How the Interpreter Works:


MAIN LOOP
Increment the PC, leaving it containing the address of the first byte after the opcode.
Initiate a fetch of the next byte into MBR. This byte will always be needed sooner or later, either as an operand for the current IJVM instruction or as the next opcode
Perform a multiway branch to the address contained in MBR at the start of Main1. This address is equal to the numerical value of the opcode currently being executed. It was placed there by the previous microinstruction.
MAIN LOOP

The fetch of the next byte is started here so it will be available by the start of the third microinstruction. It may or may not be needed then, but it will be needed eventually, so starting the fetch now cannot do any harm in any case.

It is up to the microassembler to place each microinstruction at a suitable address and link them together in short sequences using the NEXT ADDRESS field. Each sequence starts at the address corresponding to the numerical value of the IJVM opcode it interprets (e.g., POP starts at 0x57), but the rest of the sequence can be anywhere in the control store, and not necessarily at consecutive addresses.

For Example, for an IADD Operation:
The microinstruction branched to by the main loop is the one labeled iadd1. This instruction starts the work specific to IADD:

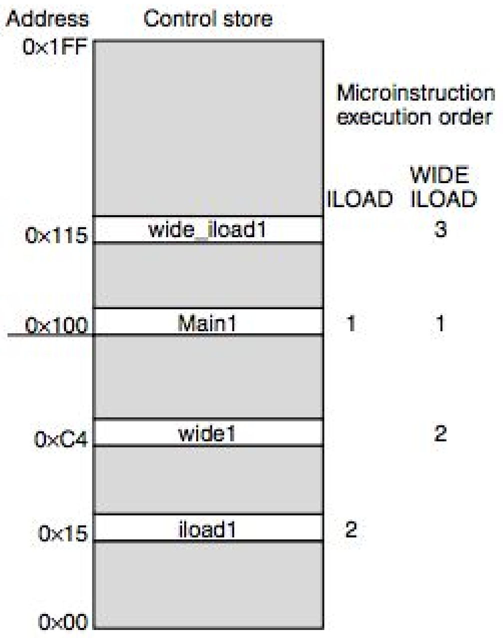
1. TOS is already present, but the next-to-top word of the stack must be fetched from memory.
2. TOS must be added to the next-to-top word fetched from memory.
3. The result, which is to be pushed on the stack, must be stored back into memory, as well as stored in the TOS register.

In order to fetch the operand from memory, it is necessary to decrement the stack pointer and write it into MAR. Note that, conveniently, this address is also the address that will be used for the subsequent write. Furthermore, since this location will be the new top of stack, SP should be assigned this value. **Therefore, a single operation can determine the new value of SP and MAR, decrement SP, and write it into both registers.**

These things are accomplished in the first cycle, iadd1, and the read operation is initiated. In addition, MPC gets the value from iadd1's NEXT ADDRESS field, which is the address of iadd2, wherever it may be. Then iadd2 is read from the control store. During the second cycle, while waiting for the operand to be read in from memory, **we copy the top word of the stack from TOS into H, where it will be available for the addition when the read completes.**

At the beginning of the third cycle, iadd3, MDR contains the addend fetched from memory. In this cycle it is added to the contents of H, and the result is stored back to MDR, as well as back into TOS. A write operation is also initiated, storing the new top-of-stack word back into memory. **In this cycle the goto has the effect of assigning the address of Main1 to MPC, returning us to the starting point for the execution of the next instruction.**

Understanding the Microinstruction Sequence Picture



What the pope is happening here? I don't know either - let's find out using ILOAD as an example.

Both ILOAD and ISTORE are restricted in that they can access only the first 256 local variables. While for most programs this may be all the local variable space needed, it is, of course, necessary to be able to access a variable wherever it is located in the local variable space.

To achieve this, IJVM uses the same mechanism employed in JVM to achieve this: a special opcode WIDE, known as a prefix byte, followed by the ILOAD or ISTORE opcode. When this sequence occurs, the definitions of ILOAD and ISTORE are modified, with a 16-bit index following the opcode rather than an 8-bit index. This is shown below.

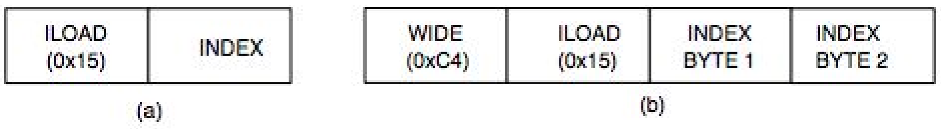


Figure 4-19. (a) ILOAD with a 1-byte index. (b) WIDE ILOAD with a 2-byte index.

WIDE is decoded in the usual way, leading to a branch to wide1 which handles the WIDE opcode. Although the opcode to widen is already available in MBR, wide1 fetches the first byte after the opcode as per microprogram logic.

Then a second branch is thus done in wide2, this time using the byte following WIDE for choosing of an action. However, since WIDE ILOAD requires different microcode than ILOAD, and WIDE ISTORE requires different microcode than ISTORE, etc, the second branch cannot just use the opcode as the target address, the way Main1 does.

Instead, wide2 ORs 0x100 with the opcode while putting it into MPC. Why? I don't know.

As a result, the interpretation of WIDE ILOAD starts at 0x115 (instead of 0x15), the interpretation of WIDE ISTORE starts at 0x136 (instead of 0x36), and so on. In this way, every WIDE opcode starts at an address 256 (i.e., 0x100) words higher in the control store higher than the corresponding regular opcode. The initial sequence of microinstructions for both ILOAD and WIDE ILOAD is shown above.

4.4. Design of the Microarchitecture Level

1. Speed vs Cost

Speed improvements due to organization and optimisation, while less amazing than that due to faster circuits, have recently been impressive. Speed can be measured in a variety of ways, but given a circuit technology and an **ISA**, there are three basic approaches for increasing the speed of execution. Each of these approaches has a specific cost, which must always be considered when making decisions on design and execution.

1. **Reducing clock cycles needed to execute an instruction (Path Length)**
   We'll be discussing this more in depth next, however for the discussion of Speed vs Cost, we'll introduce it here. Sometimes the path length can be shortened by adding specialized hardware. For example, by adding an incrementer (conceptually, an adder with one side permanently wired to add 1) to our Program Counter, we no longer have to use the **ALU** to advance the **PC**, eliminating cycles.

   The price paid for more speed is therefore more hardware. However, this capability does not help as much as might be expected. For most instructions, the cycles consumed incrementing the **PC** are also cycles where a read operation is being performed. The subsequent instruction could not be executed earlier anyway because it depends on the data coming from the memory, thus making this irrelevant.

2. **Simplify the organisation so that the clock cycle can be shorter**
3. **Overlap the execution of instructions**
   To solve this problem, separating out the circuitry for fetching the instructions—the 8-bit memory port, and the **MBR** and **PC** registers—is most effective if the unit is made functionally independent of the main data path. In this way, it can fetch the next opcode or operand on its own, independently from the current clock cycle, potentially asynchronously with respect to the rest of the **CPU** and fetching one or more instructions ahead.

   The price paid would be the need for more registers to store data temporarily whilst waiting for the current instruction to finish.

The first two are obvious, but there is a surprising variety of design opportunities that can dramatically affect either the number of clock cycles, the clock period or most often, both. In this section, we will give an example of how the encoding and decoding of an operation can affect the clock cycle.

Of course, speed is only half the picture. Cost is the other half. Cost can also be measured in a variety of ways, but a precise definition of cost is problematic. Some measures are as simple as a count of the number of components. This was particularly true in the days when processors were built of discrete components that were purchased and assembled. Today, the entire processor exists on a single chip, but bigger, more complex chips are much more expensive than smaller, simpler ones.

Individual components—for example, transistors, gates, or functional units—can be counted, but often the count is not as important as the amount of area required on the integrated circuit. The more area required for the functions included, the larger the chip. And the manufacturing cost of the chip grows much faster than its area. For this reason, designers often speak of cost in terms of ‘‘real estate,’’ that is, the area required for a circuit (presumably measured in pico-acres).

For Example: One of the most thoroughly studied circuits in history is the binary adder. There have been thousands of designs, and the fastest ones are much quicker than the slowest ones. They are also far more complex. The system designer has to decide whether the greater speed is worth the real estate.

2. Reducing the Execution Path Length

The Mic-1 was designed to be both moderately simple and moderately fast.

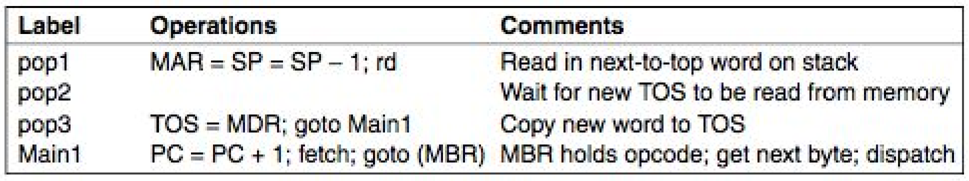
The Mic-1 **CPU** also uses a minimum amount of hardware: 10 registers, the simple **ALU** repeated 32 times (32-bit), a shifter, a decoder, a control store, and a bit of prestik. The whole system could be built with fewer than 5000 transistors plus whatever the control store (ROM) and main memory (RAM) take.

How can we improve our design for a faster solution?

Merging the Interpreter Loop with the Microcode

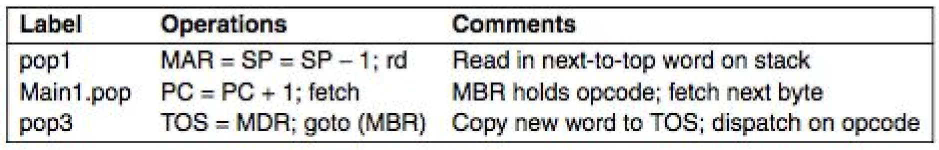
In the Mic-1, the main loop consists of one microinstruction that must be executed at the beginning of every **IJVM** instruction. In some cases it is possible to overlap it with the previous instruction. In fact, this has already been partially accomplished. Notice that when Main1 is executed, the opcode to be interpreted is already in **MBR**. It is there because it was fetched either by the previous main loop (if the previous instruction had no operands) or during the execution of the previous instruction.

The **POP** instruction is particularly well suited for this treatment, because it has a dead cycle in the middle that does not use the **ALU**. The main loop, however, does use the **ALU**. See how a pop instruction can be optimised below:



Label	Operations	Comments
pop1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-23. Original microprogram sequence for executing POP.



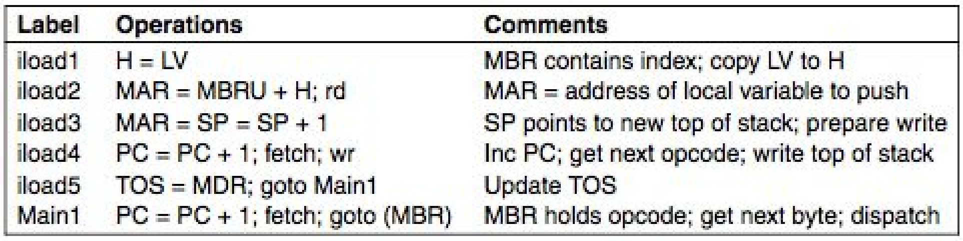
Label	Operations	Comments
pop1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
Main1.pop	PC = PC + 1; fetch	MBR holds opcode; fetch next byte
pop3	TOS = MDR; goto (MBR)	Copy new word to TOS; dispatch on opcode

Figure 4-24. Enhanced microprogram sequence for executing POP.

A Three-Bus Architecture

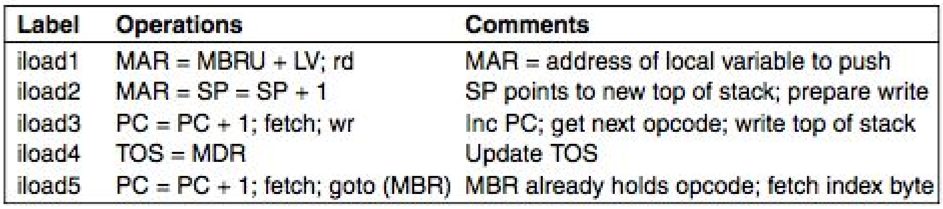
Another easy fix is to have two full input buses to the **ALU**: **A** bus and a **B** bus. This gives three buses in total instead of merely our **B** bus. All (or at least most) of the registers should have access to both input buses. The advantage of having two input buses is that it then becomes possible to add any register to any other register in one cycle.

The **ILOAD** instruction is well suited to this optimisation, as can be seen in the two implementations below:



Label	Operations	Comments
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-25. Mic-1 code for executing ILOAD.



Label	Operations	Comments
iload1	MAR = MBRU + LV; rd	MAR = address of local variable to push
iload2	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload3	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload4	TOS = MDR	Update TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR already holds opcode; fetch index byte

Figure 4-26. Three-bus code for executing ILOAD.

An Instruction Fetch Unit

In order to get dramatic speed increases, let us step back and look at the common parts of every instruction: the fetching and decoding of the fields of the instruction. Notice that for every instruction the following operations may occur:

1. The **PC** is passed through the **ALU** and incremented.
2. The **PC** is used to fetch the next byte in the instruction stream.
3. Operands are read from memory.
4. Operands are written to memory.
5. The **ALU** does a computation and the results are stored back.

If an instruction has additional fields (i.e. operands from memory), each field must be explicitly fetched, 1 byte at a time, and assembled before it can be used. This process takes **CPU** clock cycles, in fact, at least one cycle per byte to increment the **PC**, and then more again to assemble the resulting index or offset. The **ALU** is therefore used nearly every cycle for a variety of operations having to do with fetching the instruction and assembling the fields within the instruction, all before the actual work of the instruction can take place.

We can free the **ALU** from some of these tasks by dedicating a chip to fetching and assembling of operands for use by the **ALU**. This task is given to the **Instruction Fetch Unit (or IFU).** This unit can independently increment **PC** and fetch bytes from the byte stream before they are needed.

This is done in one of two possible ways,

1. The **IFU** can actually interpret each opcode, determining how many additional fields must be fetched, and assemble them into a register ready for use by the main execution unit.
2. The **IFU** can take advantage of the stream nature of the instructions and make available at all times the next 8- and 16-bit pieces, whether or not doing so makes any sense. The main execution unit can then ask for whatever it needs.

This second way is detailed in the diagram below

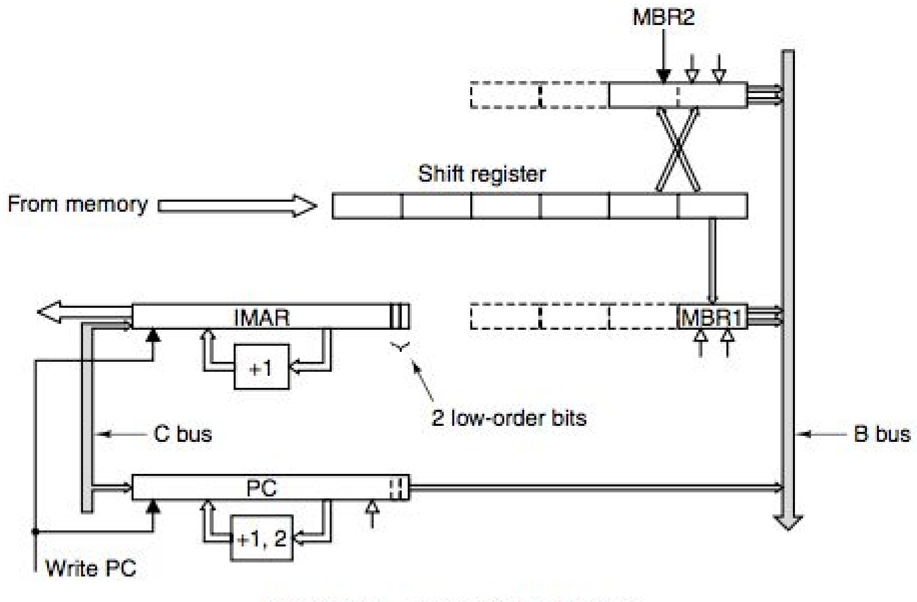


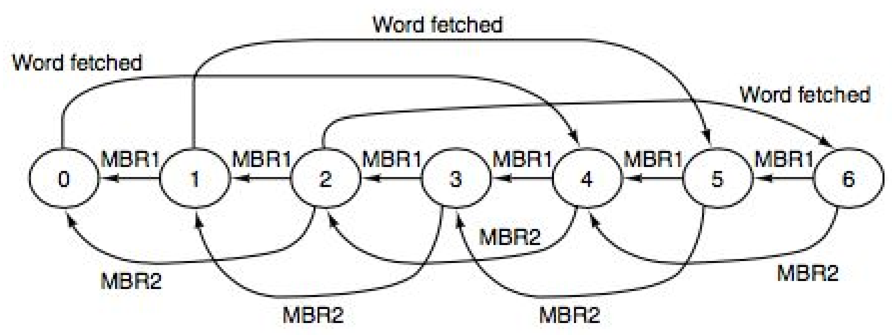
Figure 4-27. A fetch unit for the Mic-1.

Rather than a single 8-bit MBR, there are now two MBRs: the 8-bit MBR1 and the 16-bit MBR2. The IFU keeps track of the most recent byte or bytes consumed by the main execution unit. It also makes available in MBR1 the next byte, just as in the Mic-1, except that it automatically senses when the MBR1 is read, prefetches the next byte, and loads it into MBR1 immediately.

Similarly, MBR2 provides the same functionality but holds the next 2 bytes. It also has two interfaces to the B bus: MBR2 and MBR2U, gating the 32-bit sign-extended and zero-extended values, respectively.

Whenever MBR1 is read, the shift register shifts right 1 byte. Whenever MBR2 is read, it shifts right 2 bytes. Then MBR1 and MBR2 are reloaded from the oldest byte and pair of bytes, respectively. If there is sufficient room now left in the shift register for another whole word, the IFU starts a memory cycle in order to read it.

Below is a FSM (Finite State Machine), with numbers corresponding to the current number of bytes in the shift register. **Each arc represents an event.**



The first event is 1 byte being read from MBR1. This event causes the shift register to be activated and 1 byte shifted off the right-hand end, reducing the state by 1.

The second event is 2 bytes being read from MBR2, which reduces the state by two. Both of these transitions cause MBR1 and MBR2 to be reloaded. When the FSM moves into states 0, 1, or 2, a memory reference is started to fetch a new word (assuming that the memory is not already busy reading a word). The arrival of the word advances the state by 4.

To work correctly, the IFU must block when it is asked to do something it can- not do, such as supply the value of MBR2 when there is only 1 byte in the shift register and the memory is still busy fetching a new word. Also, it can do only one thing at a time.

Finally, whenever PC is changed, the IFU must be updated. Such details make it more complicated than we have shown. Still, many hardware devices are constructed as FSMs.

3. A Design with Prefetching - Mic-2

Mic-2 is defined as a machine that includes an IFU (Instruction Fetch Unit) as well as a three-bus system.

The Instruction Fetch Unit (as shown above) can greatly reduce the path length of the average instruction.

1. First, it eliminates the main loop entirely, since the end of each instruction simply branches directly to the next instruction.
2. Second, it avoids tying up the ALU incrementing PC.
3. Third, it reduces the path length whenever a 16-bit index or offset is calculated, because it assembles the 16-bit value and supplies it directly to the ALU as a 32-bit value, avoiding the need for assembly in H.

The diagram below shows our new machine (Mic-2) with the IFU added, as well as with dedicated buses A & B.

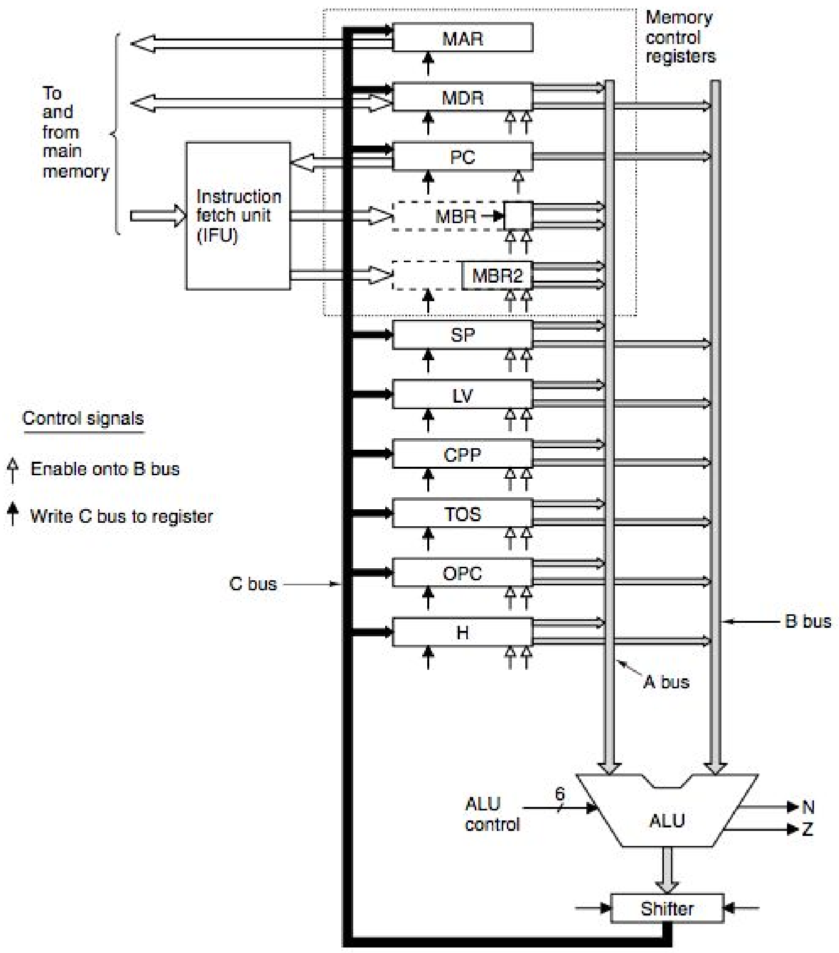


Figure 4-29. The data path for Mic-2.

As an example of how the Mic-2 works, look at IADD. It fetches the second word on the stack and does the addition as before, only now it does not have to go to Main1 when it is done to increment PC and dispatch to the next microinstruction. When the IFU sees that MBR1 has been referenced in iadd3, its internal shift register pushes everything to the right and reloads MBR1 and MBR2.

It then does a state change as per the state-diagram above. If the new state is 2, the IFU starts fetching a word from memory. All of this is in hardware. The microprogram does not have to do anything. That is why IADD can be reduced from four microinstructions to three.

4. A Pipelined Design - Mic-3

The Mic-2 is clearly an improvement over the Mic-1. It is faster and uses less control store, although the cost of the IFU will undoubtedly more than what we won by having a smaller control store. Thus it is a considerably faster machine at a marginally higher price.

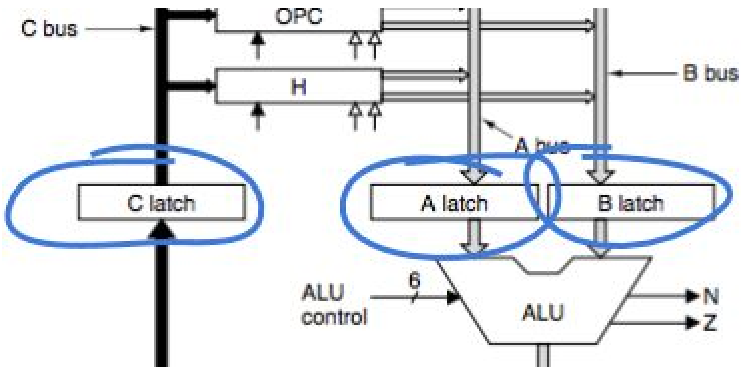
Now, we can focus on improvement by implementing a pipelined design, thereby decreasing our cycle time. This means that we can ultimately have a higher clock speed!

At the moment, the Mic-2 is very sequential. It puts registers onto its buses, waits for the ALU and shifter to process them, and then writes the results back to the registers. Except for the IFU, little parallelism is present. Adding parallelism is a real opportunity.

There are three major components to the actual data path cycle:

1. The time to drive the selected registers onto the A and B buses.
2. The time for the ALU and shifter to do their work.
3. The time for the results to get back to the registers and be stored.

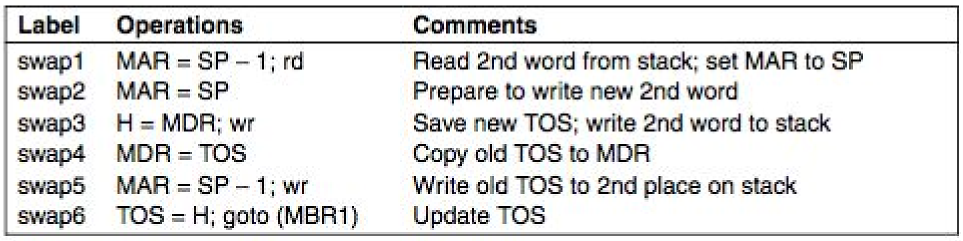
In order to add this parallel processing, we introduce a new three-bus architecture, including the IFU, but with three additional latches (registers), one inserted in the middle of each bus. The latches are written on every cycle. In effect, these registers partition the data path into distinct parts that can now operate independently of one another. We will refer to this as Mic-3, or the pipelined model.



Contrary to logic, these latches do **not** add extra cycles on for fetching and storing. Instead, by breaking up the data path into three parts, we reduce the maximum delay with the result that the clock frequency can be higher. Thus, from above, selecting registers onto the A and B buses **(1)**, waiting for the ALU to do its work **(2)** and putting results back to the registers **(3)** can all happen at once!

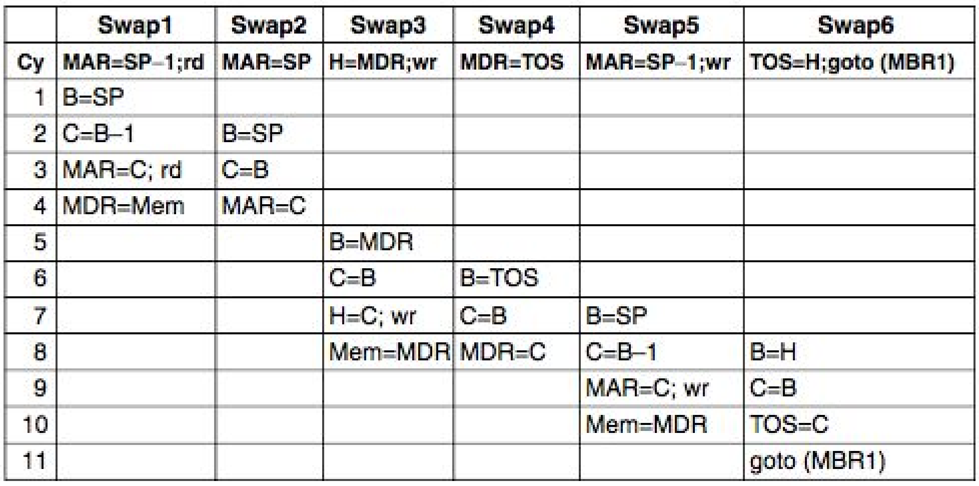
Let us suppose that by breaking the data path cycle into three time intervals, each one is about 1/3 as long as the original all happening in parallel. Thus, we can effectively triple the clock speed. (This is not totally realistic, but oh well, it's magic - you know(_ow_ow)).

This can be seen for the SWAP operation, as detailed below:



Label	Operations	Comments
swap1	MAR = SP – 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS

Figure 4-32. The Mic-2 code for SWAP.



	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cy	**MAR=SP–1;rd**	**MAR=SP**	**H=MDR;wr**	**MDR=TOS**	**MAR=SP–1;wr**	**TOS=H;goto (MBR1)**
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figure 4-33. The implementation of SWAP on the Mic-3.

As you can see, our Mic-3 program requires more cycles, it’s clock speed is much higher and is therefore far faster.

If we call the Mic-3 cycle time ΔT nsec, then the Mic-3 requires $11\Delta T$ nsec to execute SWAP. In contrast, the Mic-2 takes 6 cycles at $3\Delta T$ each, for a total of $18\Delta T$.

Pipelining has made the machine faster, even though we had to stall once to avoid a dependence.

4. A Seven-Stage Pipeline - Mic-4

The idea of pipelining parts of our cycle can be extended much further. So far, one point we have glossed over is that every microinstruction selects its own successor. Most of them just select the next one in the current sequence, but the last one, such as swap6, often does a multiway branch, which gums up the pipeline since continuing to prefetch after it is impossible. We need a better way of dealing with this point.

To solve this, we introduce a few new stages to our pipeline. Our last microarchitecture is therefore the Mic-4. Like the Mic-3, it has an IFU that prefetches words from memory and maintains the various MBRs.

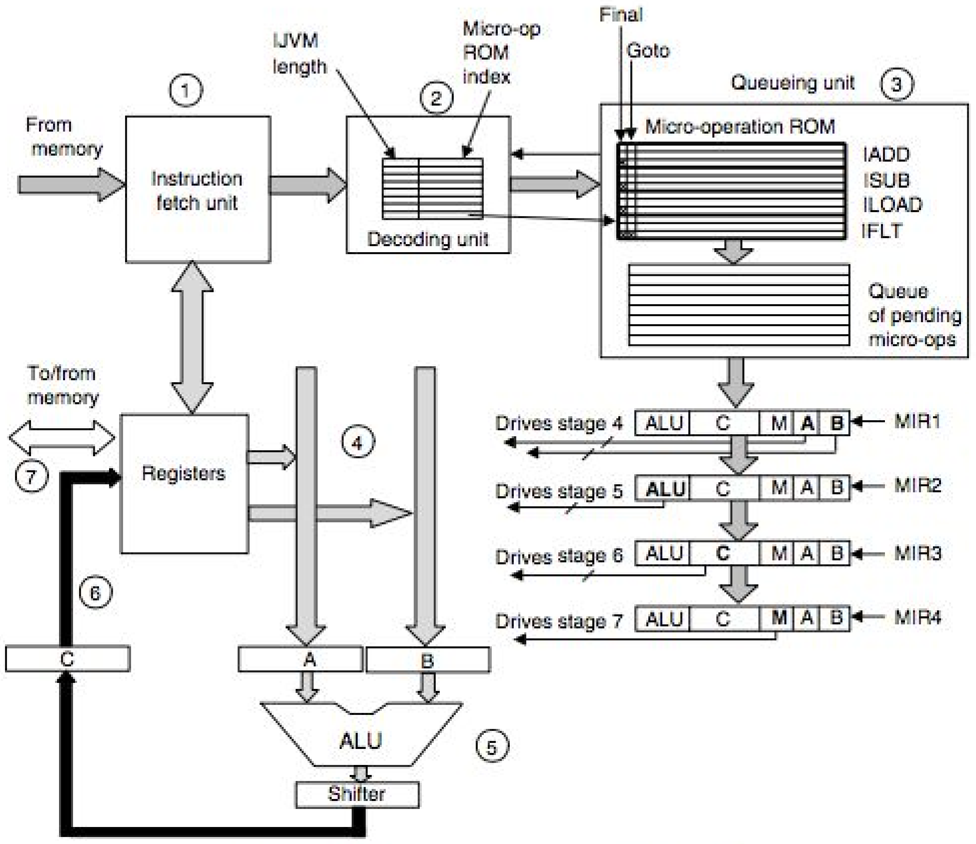


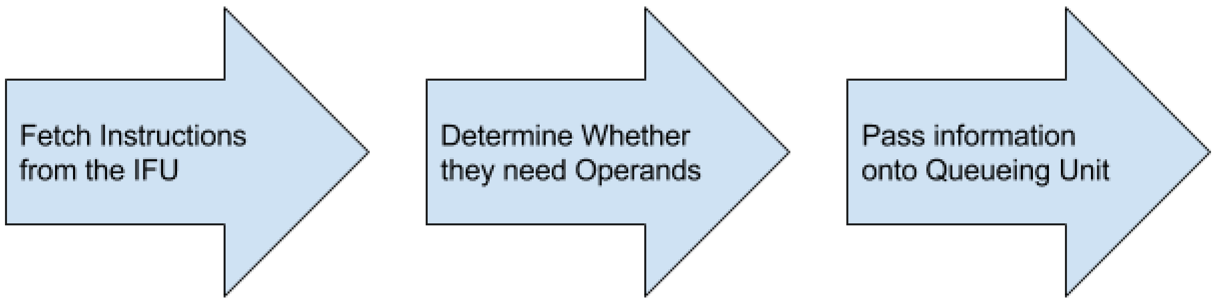
Figure 4-35. The main components of the Mic-4.

The Decoding Unit

The Instruction Fetch Unit also feeds the incoming byte stream to a new component, the **decoding unit**. This unit has an internal ROM indexed by IJVM opcode. Each entry in this ROM contains two parts: the length of that IJVM instruction and an index into another ROM, the micro-operation ROM.

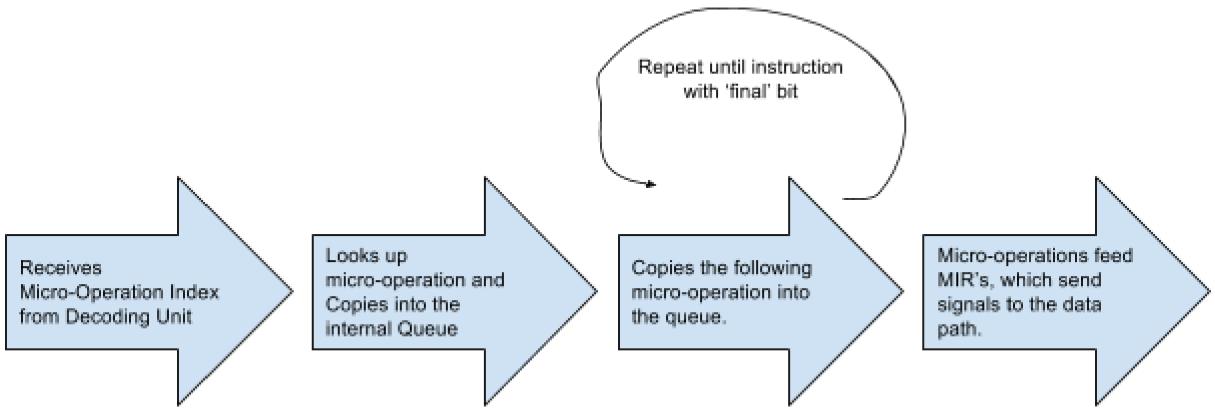
The IJVM instruction length is used to allow the decoding unit to parse the incoming byte stream into instructions, so it always knows which bytes are opcodes and which are operands. If the current instruction length is 1 byte (e.g., POP), then the decoding unit knows that the next byte is an opcode. If, however, the current instruction length is 2 bytes, the decoding unit knows that the next byte is an operand, followed immediately by another opcode!

A special case is when the WIDE prefix is seen, the following byte is transformed into a special wide opcode, for example, WIDE + ILOAD becomes WIDE ILOAD.


Fetch Instructions from the IFU
Determine Whether they need Operands
Pass information onto Queueing Unit

The Queueing Unit

The decoding unit ships the index into the queueing unit. This unit contains some logic plus two internal tables, one in ROM and one in RAM. The ROM contains the microprogram, with each IJVM instruction having some number of consecutive entries, called micro-operations. These entries must be in order, and so each IJVM sequence must be spelled out in full, duplicating sequences in some cases


Repeat until instruction with 'final' bit
Receives Micro-Operation Index from Decoding Unit
Looks up micro-operation and Copies into the internal Queue
Copies the following micro-operation into the queue.
Micro-operations feed MIR's, which send signals to the data path.

To make this work properly, we have introduced four independent MIRs. At the start of each clock cycle, MIR3 is copied to MIR4, MIR2 is copied to MIR3, MIR1 is copied to MIR2, and MIR1 is loaded with a fresh micro-operation from the micro-operation queue. Then each MIR puts out its control signals, but only some of them are used. The A and B fields from MIR1 are used to select the registers that drive the A and B latches, but the ALU field in MIR1 is not used and is not connected to anything else in the data path.

This means that, one after the other, our MIRs are read to operate different stages of our pipeline. Ultimately, this leads us to the following Mic-4 pipeline design, of seven stages.

Our Design

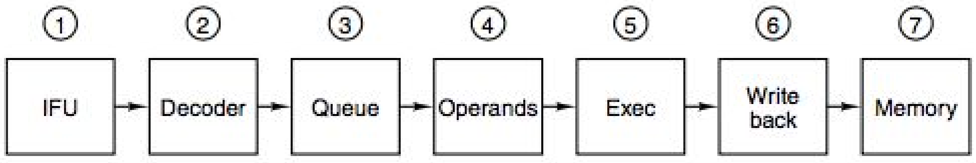


Figure 4-36. The Mic-4 pipeline.

On each clock cycle, the MIRs are shifted forward and the micro operation at the bottom of the queue is copied into MIR1 to start its execution. The control signals from the four MIRs then spread out through the data path, causing actions to occur. Each MIR controls a different portion of the data path and thus different microsteps.

In this design we have a deeply pipelined CPU, which allows the individual steps to be very short and thus the clock frequency high. Many CPUs are designed in essentially this way, especially those that have to implement an older (CISC) instruction set. For example, the Core i7 implementation is conceptually similar to the Mic-4 in some ways.

4.5. Improving Performance* *(The one with a lot of reading)

All computer manufacturers want their systems to run as fast as possible. In this section, we will look at a number of advanced techniques currently being investigated to improve system (primarily CPU and memory) performance. Due to the highly competitive nature of the computer industry, the lag between new research ideas that can make a computer faster and their incorporation into products is surprisingly short. Consequently, most of the ideas we will discuss are already in use in a wide variety of existing products.

These can be categorized into roughly two categories:

Implementation Improvements
Ways of building a new CPU or memory to make the system run faster without changing the architecture.

Architectural Improvements
One way to improve speed is through a faster clock. However, changing the architecture of our machine removes backwards compatibility.

1. Cache Memory

The Problem

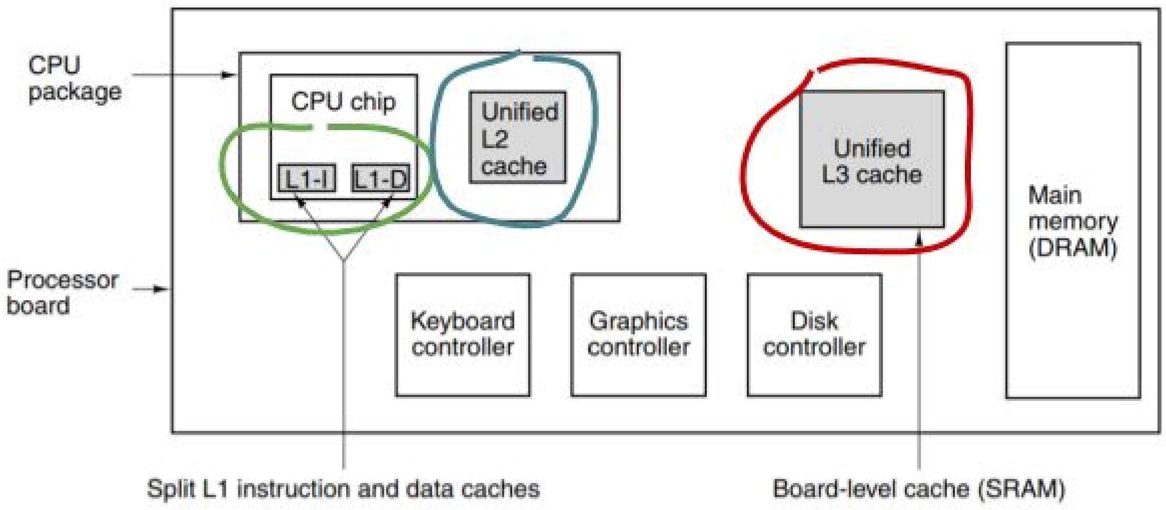
One of the most challenging aspects of computer design has been to provide a memory system that responds with the same speed as the CPU requesting it. This is nearly impossible due to the increasing speeds of CPU designs.

Modern processors place overwhelming demands on memory systems, in terms of both latency and bandwidth. Unfortunately, most of the time, increasing bandwidth also results in increasing latency. These counteract each other. Thus, we must introduce a system for storing data in a very-fast and easily accessible location. This is known as **cache.**

Cache is a very fast storage unit which is used to pre-load data from main memory for use by the CPU. This runs at almost the same speed as a CPU, however not as fast. It is, however, much faster than a main memory call.

One of the most effective ways to improve both bandwidth and latency is to use multiple caches, each dedicated to storing different data. There are several benefits from having separate caches for different data, namely instructions and data. This is called a **split-cache. Unified Cache, however, is where these different data are stored on a single cache.**

Today, many systems are far more complicated than this, and add additional caching systems. This is known as level 2 cache, which may reside between the instruction and data caches, and main memory. This can be expanded to include level three caching (adding a third cache module) etc. You can see this type of system below.



The green cache above, L1-I and L1-D, are the split level one instruction and data caches. These are incredibly fast and operate at the same speed as the CPU.

The blue cache above, Unified L2 Cache, lies on the CPU package, but not on the CPU chip itself. This is connected directly to the CPU through a high-speed dedicated path/bus, and is thus relatively fast. Data is usually loaded into the L2 cache for use later by the CPU.

The red cache above, Unified L3 cache, lies off the CPU package, and its between the main memory and the CPU. This cache is faster than main memory, however slower than L1 and L2 cache.

Cache is usually inclusive, with the full contents of L1 cache being available in L2, and the full contents of L2 cache being available in L3.

Address Locality

To achieve their goal, caches depend on two kinds of address locality

1. **Spatial Locality**
   The observation that memory locations with addresses numerically similar (memory close together) to a recently accessed memory location are likely to be accessed in the near future. Caches try to exploit this property by bringing in more data than have been requested, with the expectation that future requests can be anticipated.

2. **Temporal locality**
   Occurs when recently accessed memory locations are accessed again. This may occur, for example, to memory locations near the top of the stack, or instructions inside a loop. Temporal locality is exploited in cache designs primarily by the choice of what to discard on a cache miss. Many cache replacement algorithms exploit temporal locality by discarding those entries that have not been recently accessed.

Cache Lines

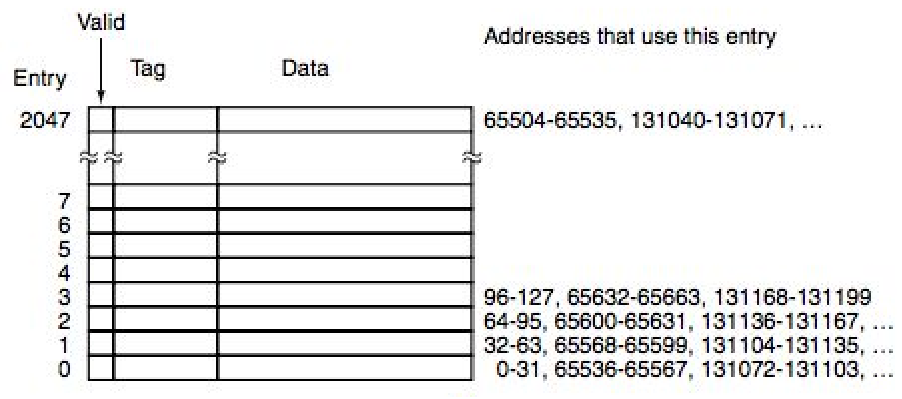
All caches use the following model. Main memory is divided up into fixed-size blocks called **cache lines**. A cache line typically consists of 4 to 64 consecutive bytes. Lines are numbered consecutively starting at 0, so with a 32-byte line size, line 0 is bytes 0 to 31, line 1 is bytes 32 to 63, and so on.

At any instant, some lines are in the cache. When memory is referenced, the cache controller circuit checks to see if the word referenced is currently in the cache. If so, the value there can be used, saving a trip to main memory. If the word is not there, some unused line entry is removed from the cache and the line needed is fetched from memory or more distant cache to replace it.

Many variations on this scheme exist, but in all of them the idea is to keep the most heavily used lines in the cache as much as pos- sible, to maximize the number of memory references satisfied out of the cache.

Direct-Mapped Caches

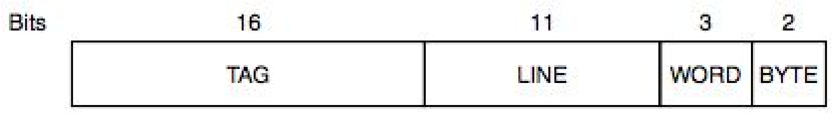
The simplest cache is known as a direct-mapped cache. An example single-level direct-mapped cache is shown below. This example cache contains 2048 entries. Each entry (row) in the cache can hold exactly one cache line from main memory. With a 32-byte cache line size (for this example), the cache can hold 2048 entries of 32 bytes or 64 KB in total.



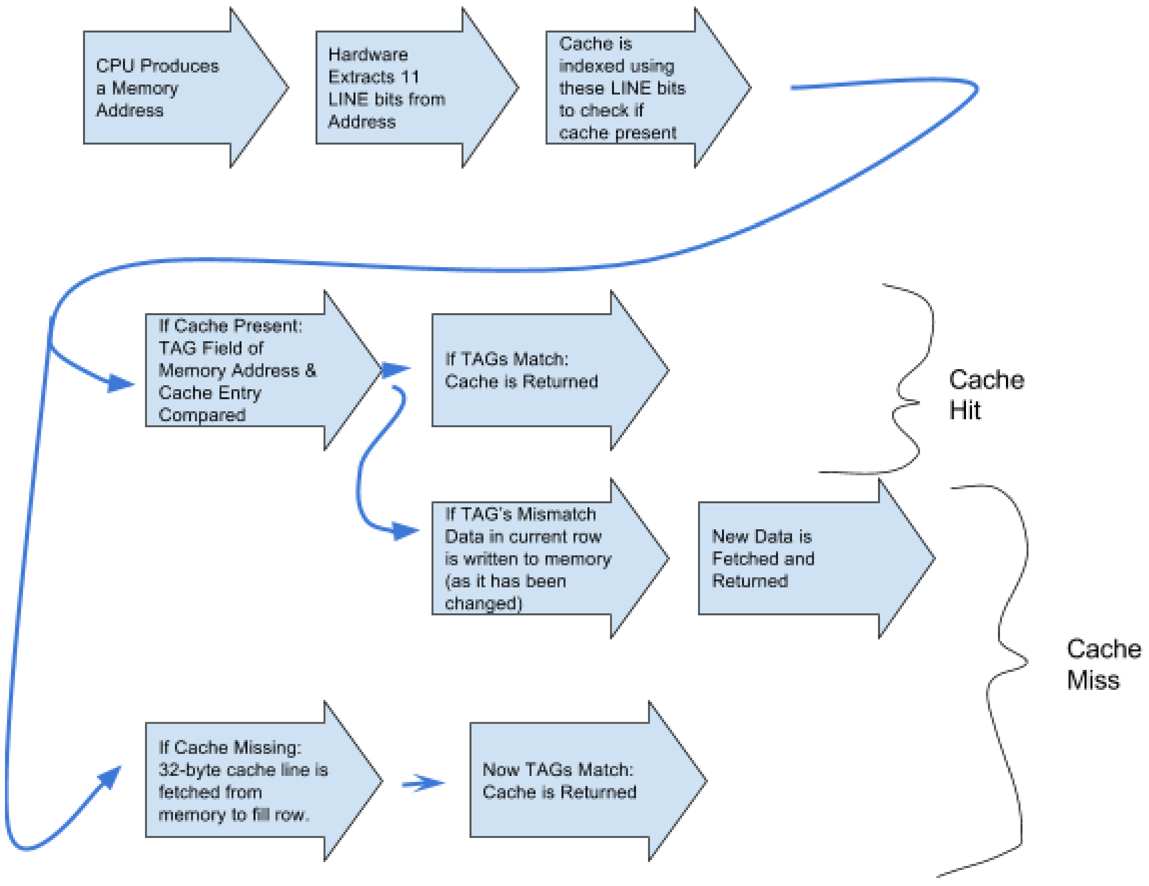
Each cache entry consists of three parts:

1. The Valid bit indicates whether there is any valid data in this entry or not. When the system is booted (started), all entries are marked as invalid.
2. The Tag field consists of a unique, 16-bit value identifying the corresponding line of memory from which the data came.
3. The Data field contains a copy of the data in memory. This field holds one cache line of 32 bytes.

In a direct-mapped cache, a given memory word can be stored in exactly one place within the cache. Given a memory address, there is only one place to look for it in the cache. If it is not there, then it is not in the cache. For storing and retrieving data from the cache, the address is broken into four components as below:



1. The TAG field corresponds to the Tag bits stored in a cache entry.
2. The LINE field indicates which cache entry holds the corresponding data, if they are present.
   The WORD field tells which word within a line is referenced.
3. The BYTE field is usually not used, but if only a single byte is requested, it tells which byte within the word is needed. For a cache supplying only 32-bit words, this field will always be 0.


CPU Produces a Memory Address
Hardware Extracts 11 LINE bits from Address
Cache is indexed using these LINE bits to check if cache present
If Cache Present: TAG Field of Memory Address & Cache Entry Compared
If TAGs Match: Cache is Returned
If TAG's Mismatch Data in current row is written to memory (as it has been changed)
New Data is Fetched and Returned
If Cache Missing: 32-byte cache line is fetched from memory to fill row.
Now TAGs Match: Cache is Returned
Cache Hit
Cache Miss

The issue with the above is that many different addresses in memory will have the same LINE value: X and X + 65536 will have the same LINE address.

This means that cache will constantly evict existing cache out when read. We will solve this problem above.

Set-Associative Caches

As mentioned above, many different lines in memory compete for the same cache slots. If a program using the direct mapped cache above heavily uses words at addresses 0 and at 65,536, there will be constant conflicts, with each reference potentially evicting the other one from the cache.

A solution is to allow two or more lines in each cache entry. A cache with n possible entries for each address is called an n-way set-associative cache. A four-way set-associative cache is illustrated below

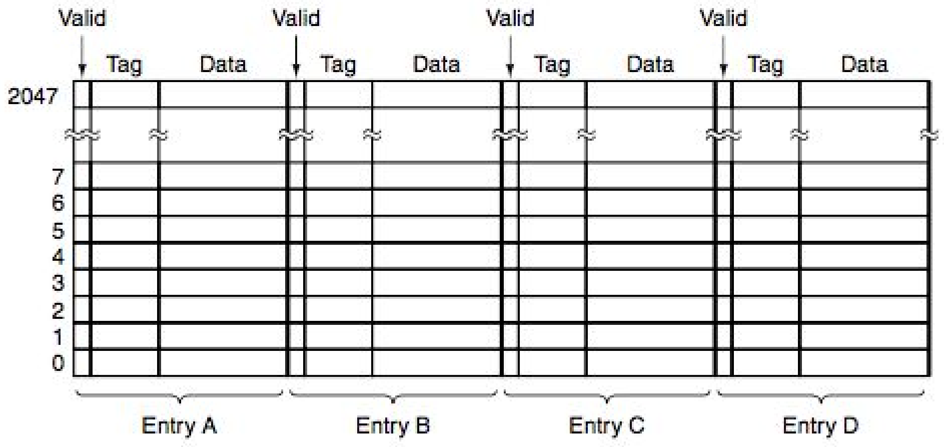


Figure 4-39. A four-way set-associative cache.

A set-associative cache is inherently more complicated than a direct-mapped cache because as a set of n cache entries must be checked to see if the needed line is present. And they have to be checked very fast. Nevertheless, experience shows that two-way and four-way caches perform well enough to make this extra circuitry worthwhile.

The use of a set-associative cache presents the designer with a choice. When a new entry is to be brought into the cache, which of the present items should be discarded? The optimal decision, of course, requires a peek into the future, but a pretty good algorithm for most purposes is **LRU (Least Recently Used)**. This is obvious, as the least recently used cache entry is probably the least-needed one.

Finally, writes pose a special problem for caches. When a processor writes a word, and the word is in the cache, it obviously must either update the word or discard the cache entry. Nearly all designs **update the cache** directly. This new cache value is then written back into main memory when the cache is being overwritten. Thus, the LRU algorithm first checks if the line it's updating contains any new data, and updates main memory accordingly.

2. Branch Prediction

The Problem

Modern computers are highly pipelined, as shown (simplified) in our seven stage pipeline of a few chapters ago.

High-end computers sometimes have 10-stage pipelines or even more. Pipelining works best on linear code, so the fetch unit can just read in consecutive words from memory and send them off to the decode unit in advance of their being needed.

The only minor problem with this wonderful model is that it is not the slightest bit realistic. Programs are not linear code sequences. They are full of branch instructions and crazy shit which makes them impossible to predict.

Why can the fetch unit not just continue to read instructions from the target address (the place that will be branched to)?

The trouble lies in the nature of pipelining. In our system, we see that instruction decoding occurs in the second stage. Thus the fetch unit has to decide where to fetch from next before it knows what kind of instruction it just got. Confusing, right? Impossibru.

Only one cycle later can it learn that it just picked up an unconditional branch, and by then it has already started to fetch the instruction following the unconditional branch. As a consequence, a substantial number of pipelined machines (such as the UltraSPARC III) have the property that the instruction following an unconditional branch is executed, even though logically it should not be. **The position after a branch is called a delay slot.**

Annoying as unconditional branches are, conditional branches are worse. Not only do they also have **delay slots**, but now the fetch unit does not know where to read from until much later in the pipeline. Early pipelined machines just stalled until it was known whether the branch would be taken or not. Stalling for three or four cycles on every conditional branch, especially if 20% of the instructions are conditional branches, wreaks havoc with the performance.

Thus, we must find a way to predict branches before they occur.

Dynamic Branch Prediction

Clearly, having the predictions be accurate is of great value, since it allows the CPU to proceed at full speed. As a consequence, much ongoing research aims at improving branch prediction algorithms.

One approach is for the CPU to maintain a history table (in special hardware), in which it logs conditional branches as they occur, so they can be looked up when they occur again. Kinda like a logbook from which we can predict future moves. The simplest version of this scheme is shown below.

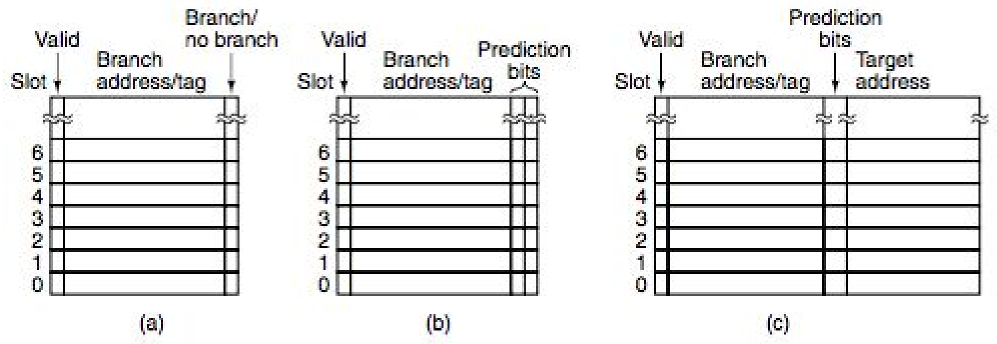


Figure 4-41. (a) A 1-bit branch history. (b) A 2-bit branch history. (c) A mapping between branch instruction address and target address.

There are several ways to organize the history table. In fact, these are precisely the same ways used to organize a cache. Consider a machine with 32-bit instructions that are word aligned so that the low-order 2 bits of each memory address are 00.

With a direct-mapped history table containing 2n entries, the low-order n + 2 bits of a branch instruction target address can be extracted and shifted right 2 bits. This n-bit number can be used as an index into the history table where a check is made to see if the address stored there matches the address of the branch.

Therefore, | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | -------> | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | (Index of 53)

Thus, each address is turned into a unique(ish) index which can be referenced from our branch table. As with a cache, there is no need to store the low-order n + 2 bits, so they can be omitted (i.e., just the upper address bits—the tag—are stored). If there is a hit, the prediction bit is used to predict the branch. If the wrong tag is present or the entry is invalid, a miss occurs, just as with a cache. In this case, the forward/backward branch rule can be used.

If the branch history table has, say, 4096 entries, then branches at addresses 0, 16384, 32768, ... will conflict, analogous to the same problem with a cache. The same solution is possible: a two-way, four-way, or n-way associative entry. As with a cache, the limiting case is a single n-way associative entry, which requires full associativity of lookup

This scheme works well in most situations. However, one systematic problem always occurs. When a loop is finally exited, the branch at the end will be mispredicted, and worse yet, the misprediction will change the bit in the history table to indicate a future prediction of ‘‘no branch.’’

The next time the loop is entered, the branch at the end of the first iteration will be predicted wrong. If the loop is inside an outer loop, or in a frequently called procedure, this error can occur often.

To eliminate this misprediction, we can give the table entry a second chance. With this method, the prediction is changed only **after two consecutive incorrect predictions**. This approach requires having two prediction bits in the history table, one for what the branch is ‘‘supposed’’ to do, and one for what it did last time

Static Branch Prediction

All of the branch prediction techniques discussed so far are dynamic, that is, are carried out at run time while the program is running. They also adapt to the program’s current behavior, which is good. The downside is that they require specialized and expensive hardware and a great deal of chip complexity.

A different way to go is to have the compiler help out. When the compiler sees a statement like:

$$for\ (i\ =\ 0;\ i\ <\ 1000000;\ i{+}{+})\ \{\ ...\ \}$$

It knows very well that the branch at the end of the loop will be taken nearly all the time $(1000000 - 1)$. Static Branch Prediction allows the compiler to determine this at runtime, and set up a system to tell the hardware what to expect.

Although this is an architectural change (and not just an implementation issue), some machines, such as the UltraSPARC III, have a second set of conditional branch instructions, in addition to the regular ones (which are needed for backward compatibility). The new ones contain a bit in which the compiler can specify that it thinks the branch will be taken (or not taken). When one of these is encountered, the fetch unit just does what it has been told. Furthermore, there is no need to waste precious space in the branch history table for these instructions, thus reducing conflicts there.

3. Out-of-Order Execution and Register Renaming

The Problem

Most modern CPUs are both pipelined and superscalar. What this generally means is that a fetch unit pulls instruction words out of memory before they are needed in order to feed a decode unit.

The decode unit issues the decoded instructions to the proper functional units for execution. In some cases it may break individual instructions into micro-ops before issuing them, depending on what the functional units can do.

Thus, it is clear to see that our machine would be simplest if the instructions were executed in the order in which they were fetch (assuming for the moment that the branch prediction algorithm never guesses wrong).

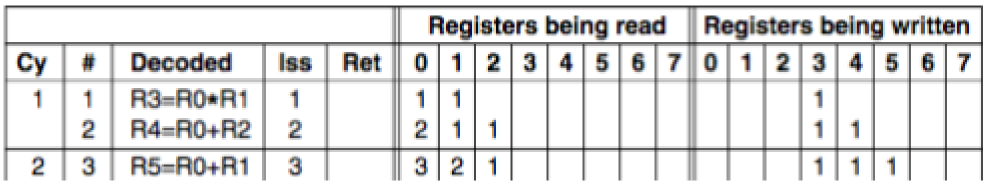
However, in-order execution does not always give optimal performance due to dependences between instructions. If an instruction needs a value computed by the previous instruction, the second one cannot begin executing until the first one has produced the needed value. In this situation **(a RAW dependence)**, the second instruction has to wait unnecessarily.

Instruction Reordering

In an attempt to get around these problems and produce better performance, some CPUs allow dependent instructions to be skipped over, to get to future instructions that are not dependent **first**. Needless to say, the internal instruction-scheduling algorithm used must deliver the same effect as if the program were executed in the order written. We will now demonstrate how instruction reordering works using a not-so-detailed example.

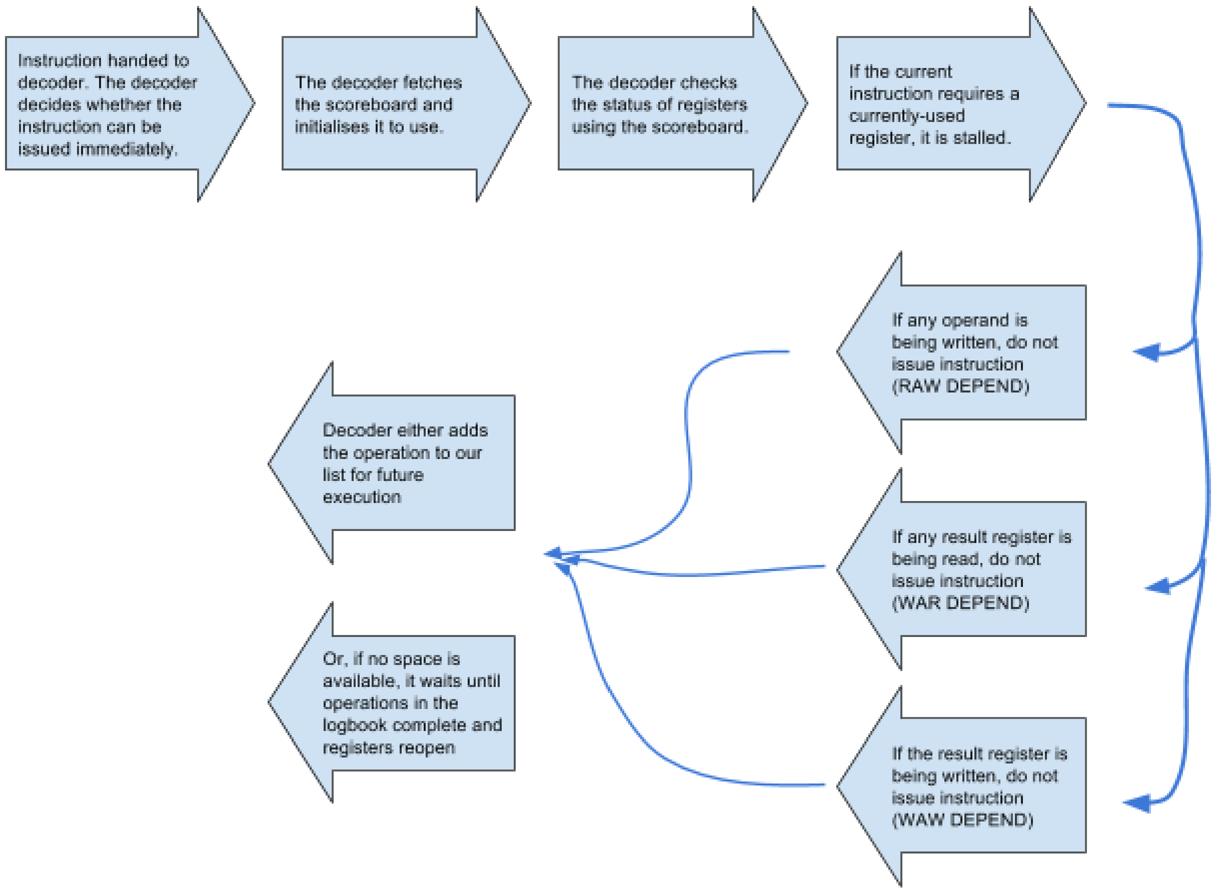
I can't summarise what's mentioned in the book. I really recommend you go and have a look on page 316. However, in essence, the following occurs:

Scoreboard:



					Registers being read								Registers being written							
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		

Before the flow diagram, the scoreboard must be explained. This CPU structure keeps track of register usage, to avoid issuing an instruction for which no registers are available for our instruction. For simplicity, we will assume there is always a suitable functional unit available, so we will not show the functional units on the scoreboard. If a maximum of, say, 15 instructions may be executing at once, then a 4-bit counter will do. When an instruction is issued, the scoreboard entries are incremented. When an instruction is retired, the entries are decremented. This means that the scoreboard can keep a 'maximum' of 15 'operations' in its logbook, which are executed only when it's absolutely vital to do so.


Instruction handed to decoder. The decoder decides whether the instruction can be issued immediately.
The decoder fetches the scoreboard and initialises it to use.
The decoder checks the status of registers using the scoreboard.
If the current instruction requires a currently-used register, it is stalled.
If any operand is being written, do not issue instruction (RAW DEPEND)
If any result register is being read, do not issue instruction (WAR DEPEND)
If the result register is being written, do not issue instruction (WAW DEPEND)
Decoder either adds the operation to our list for future execution
Or, if no space is available, it waits until operations in the logbook complete and registers reopen

We have already seen

In a **RAW** dependences, an instruction needs to use as a source a result that a previous instruction has not yet produced.

In a **WAR dependence (Write After Read),** one instruction is trying to overwrite a register that a previous instruction may not yet have finished reading.

A **WAW dependence (Write After Write)** is similar. These can often be avoided by having the second instruction put its results somewhere else (perhaps temporarily).

If none of the above three dependences exist, and the functional unit it needs is available, the instruction is issued. If not, the RAW dependency instruction is held back, whilst the other two are executed in a special fashion.

Register Renaming

We introduce a new technique for solving the problem of WAR and WAW dependencies: **register renaming**. The wise decode unit changes the use of our main registers (R0,R1,R2 etc) to something called secret registers (S0,S1,S2 etc), which are not visible to the programmer.

Now our instructions can be issued concurrently with one another, even if they use the 'same registers', as our CPU is clever enough to divert one of the instructions to another set of registers. Modern CPUs often have dozens of secret registers for use with register renaming.

On many real machines, renaming is deeply embedded in the way the registers are organized. There are many secret registers and a table that maps the registers visible to the programmer onto the secret registers. Thus the real register being used for, say, R0 is located by looking at entry 0 of this mapping table. In this way, there is no real register R0, just a binding between the name R0 and one of the secret registers. This binding changes frequently during execution to avoid dependences.

This technique often completely eliminate WAR and WAW dependences.

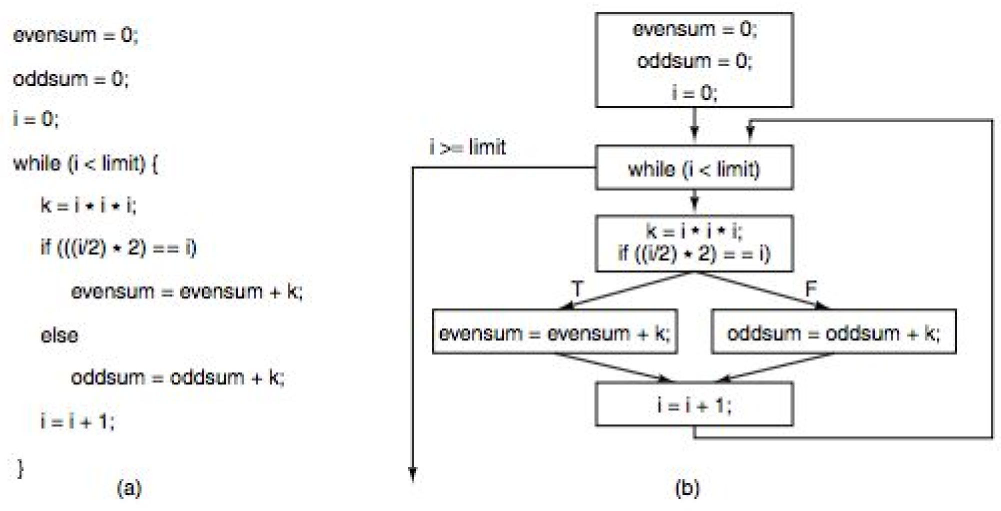
4. Speculative Execution

What is it?

Speculative execution is when code is executed before it is known whether or not we actually need the result, therefore having the result available in advance should it be needed. Whilst the benefits can be incredible should our pipelined CPU use these results, this can result in both a situation where we waste CPU cycles, as well as the potential overwriting of registers before we actually know that we can do so.

Computer programs can be broken up into basic blocks, each consisting of a linear sequence of code with one entry point on top and one exit on the bottom. A basic block does not contain any control structures (e.g., if statements or while statements) so that its translation into machine language does not contain any branches. The basic blocks are connected by control statements.

A program in this form can be represented as a directed graph, as shown below. Here we compute the sum of the cubes of the even and odd integers up to some limit and accumulate them in even sum and oddsum, respectively. Within each basic block, the reordering techniques of the previous section work fine.



```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if (((i/2) * 2) == i)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

Figure 4-45. (a) A program fragment. (b) The corresponding basic block graph.

Imagine if all the variables above were kept in registers except even-sum and oddsum (for lack of registers). It might make sense then to move their LOAD instructions to the top of the loop, before computing k, to get them started early on, so the values will be available when needed. Of course, only one of them will be needed on each iteration, so the other LOAD will be wasted, but if the cache and memory are pipelined and there are issue slots available, it might still be worth doing this. **Executing code before it is known if it is even going to be needed is called speculative execution.**

Potential Issues?

Irrevocable Results

It is essential that after executing the code, it does not cause irreversible results, in case the code was not actually needed and thus we overwrite important register values. To avoid overwriting registers before we know if this is actually needed, we use the technique of Register Renaming as detailed above to rename the destination registers for our speculative execution results.

Exception Errors

Another thing we need to watch out for is if a speculatively executed instruction causes an exception? A painful, but not fatal, example is a LOAD instruction that causes a cache miss on a machine with a large cache line size (say, 256 bytes) and a memory far slower than the CPU and cache.

If a LOAD that is actually needed stops the machine dead in its tracks for many cycles while the cache line is being loaded, well, that's life, since the word is needed and not much can be done while we wait.

However, stalling the machine to fetch a word that turns out not to be needed is counterproductive. One solution present in a number of modern machines is to have a special *SPECULATIVE-LOAD* instruction that tries to fetch the word from the cache, but if it is not there, just gives up. If the value is there when it is actually needed, it can be used, but if it is not, the hardware must go out and get it on the spot as if this system wasn't in place. If the value turns out not to be needed, no penalty has been paid for the cache miss.

Dinosaur Book: Chapter 8

This chapter deals with organizing memory hardware. These summaries will be **very** basic. Rather read the slides/book, as I might accidentally leave out crucial information.

Objectives:

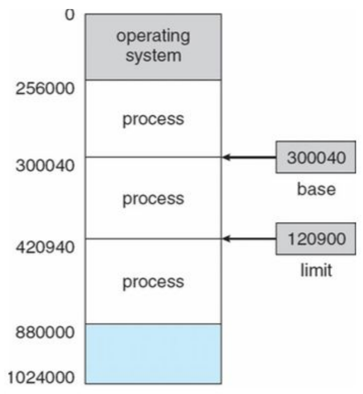
1. To provide a detailed description of various ways of organizing memory hardware
2. To discuss various memory-management techniques, including paging and segmentation
3. To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Every program is placed into main memory before it is executed. This is because **main memory and registers are the only storage units that the CPU can access.**

Cache is used in between to speed up access time. If we were to wait for main memory all the time, it would waste many cycles and stall the CPU. That's bad.

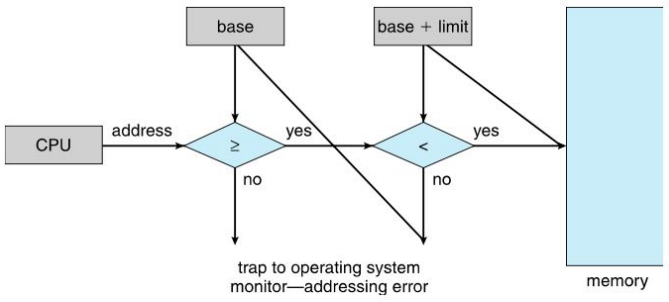
Memory unit only sees a stream of addresses + read requests, or address + data and write requests. This means that memory is dealt with in a post-box fashion, where a post person looks on their map to find where every memory element address is for fetch/delivery.

Hardware Access Protection



Protection of memory required to ensure correct operation. A pair of base and limit registers define the logical address space CPU must check every memory access generated in user mode to be sure it is between base and limit for that user. This prevents programs accessing other programs memory etc, and also prevents data leaks.

The CPU checks that the program is only accessing memory between the base and the limit.



This might seem complicated, but it's really simple. Basically, the post person can only work in his/her/their zone - and should they try to go out of their zone, they get slapped in the face with a fish and told nicely to turn around.

Address Binding

What happens?

Instead of memory always being placed into the same, inconvenient, place - address binding is used to bind memory to relocatable memory. This means that memory is accessed in a relative fashion, for example, Start of Program memory + 50 bytes etc.

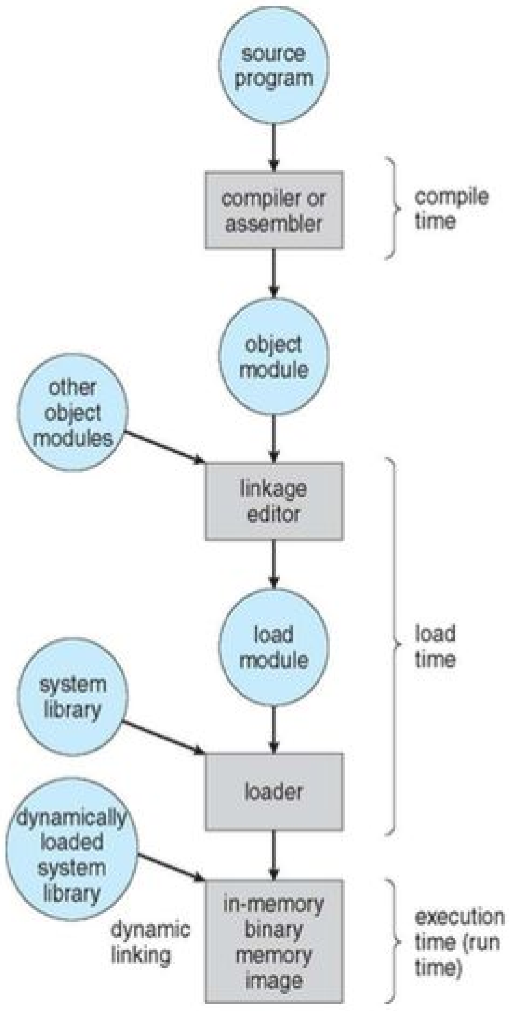
Compiled code addresses bind to relocatable addresses
    i.e. "14 bytes from beginning of this module"
Linker or loader will bind relocatable addresses to absolute addresses
    i.e. 74014

When does it happen?

Address binding of instructions and data to memory addresses can happen at three different stages

1. Compile time: If memory location known prior, absolute code can be generated; must recompile code if starting location changes
2. Load time: Must generate relocatable code if memory location is not known at compile time
3. Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
   Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing



This is supposedly self explanatory in the slides. I'll just explain a little bit.

Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Loadtime.If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work

Logical vs Physical Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address. Thus, CPU's work with logical addresses - which then are mapped to a physical address in memory.

- Logical address space is the set of all logical addresses generated by a program
- Physical address space is the set of all physical addresses generated by a program

These schemes are different, and are converted between one another using a MMU.

Memory-management Unit

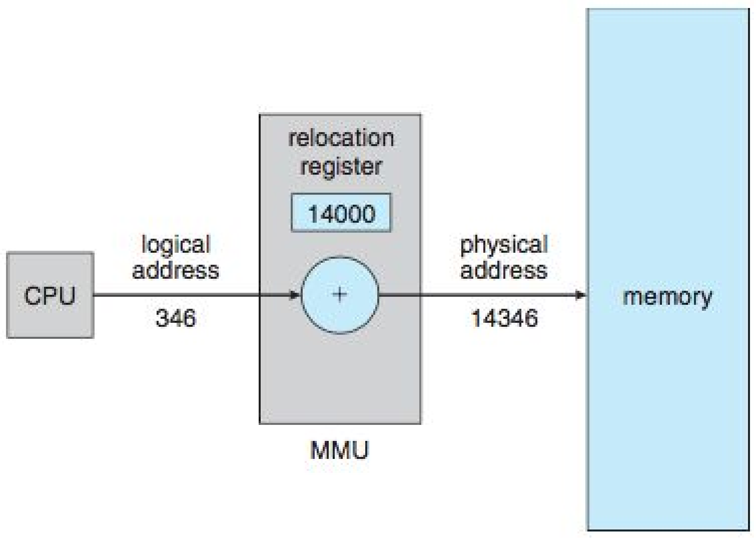


Figure 8.4 Dynamic relocation using a relocation register.

Thus, a CPU using a logical address will be sent through an MMU, which will ‘convert’ the address into a physical address in memory. This is it’s function. Many methods for doing this are possible.

To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

Base register now called relocation register
MS-DOS on Intel 80x86 used 4 relocation registers

1. The user program deals with logical addresses; it never sees the real physical addresses
2. Execution-time binding occurs when reference is made to location in memory
3. Logical address bound to physical addresses

Dynamic Linking

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run (refer back to Figure 8.3). Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.

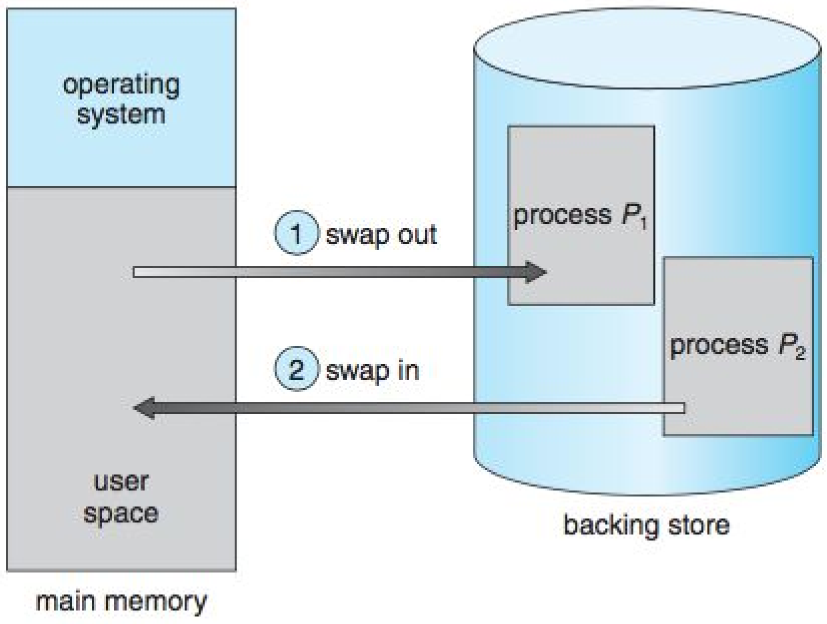
Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

A small piece of code, **called a stub**, is used to locate the appropriate memory-resident library routine. Thus, instead of including system libraries in every compiled program - the program is compiled without them, but rather with links to them. Thus, the program is first loaded into memory without libraries, and then the libraries are loaded.

These are called dependencies. Programs will not work if the dependencies are thus not installed. This is known to the system as **Shared Libraries.**

Swapping

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.



Time Calculations

Lets say a 100MB process is swapped to a hard drive that has an effective transfer rate of 50MB/s. In this process, another page is swapped back into memory (also 100MB).

Thus,
100/50 = 2s.
+ 100/50 = 2s
== 4s total swap time. This is slow af, however is needed sometimes in an emergency.

Remember, that swapping can only happen when:

- The process is idle
- The process has no I/O pending

Swapping doesn’t really happen on mobile devices, as programs are usually just halted in the background if they run into low memory. Thus, most mobile apps save state and can be reopened whenever and immediately go back to where they were. If they don't do this, they can request not to be halted.

Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

Protection

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users’ programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system’s size to change dynamically, as all it has to do is change the limit register to be larger/smaller. This flexibility is desirable in many situations. For example, loading/unloading drivers etc.

Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.

Fixed-Partition Scheme

Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple- partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Variable-Partition Scheme

In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.

Input Queue

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

Placing into Memory

This procedure is a particular instance of the general dynamic storage- allocation problem, which concerns how to satisfy a request of size n from a list of free holes.

First-Fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough

Best-Fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst-Fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

External Fragmentation
Total memory space exists to satisfy a request, but it is not contiguous (one-after-the-other)

Internal Fragmentation
Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Segmentation

In Software

As we've already seen, the user's view of memory is not the same as the actual physical memory. This is equally true of the programmer's view of memory. Indeed, dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer. What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory? The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. The programmer talks about “the stack,” “the math library,” and “the main program” without caring what addresses in memory these elements occupy.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.

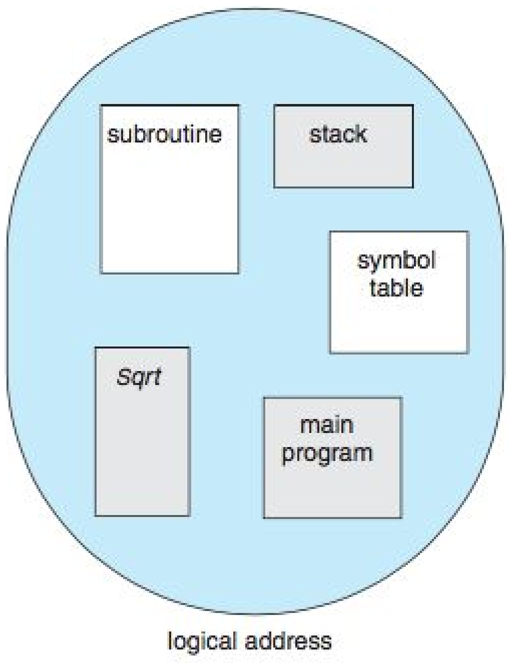


Figure 8.7 Programmer's view of a program.

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

In Hardware

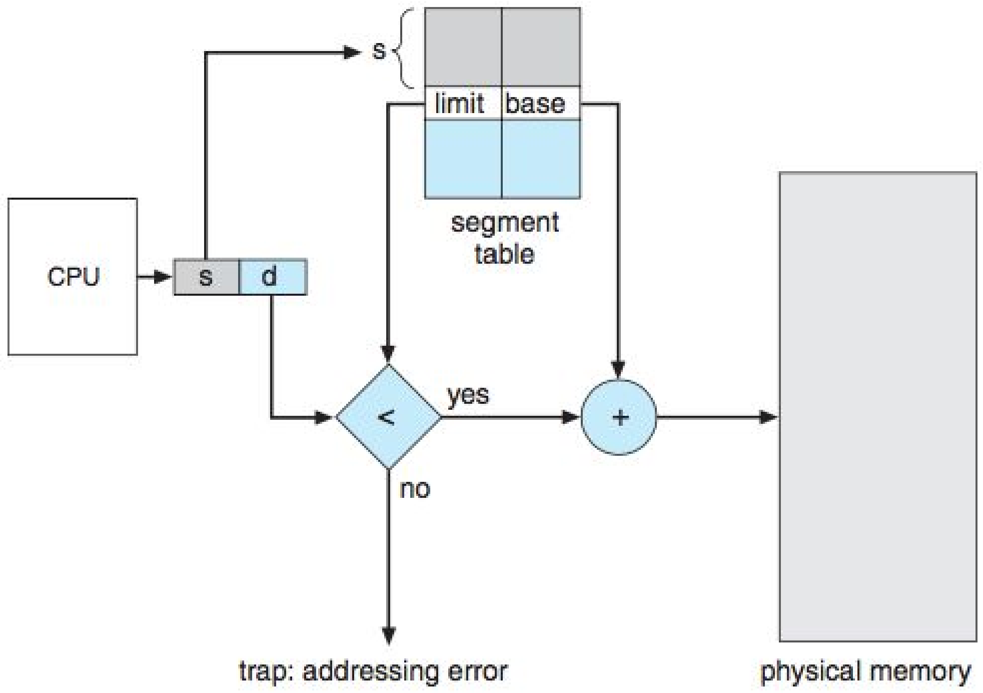
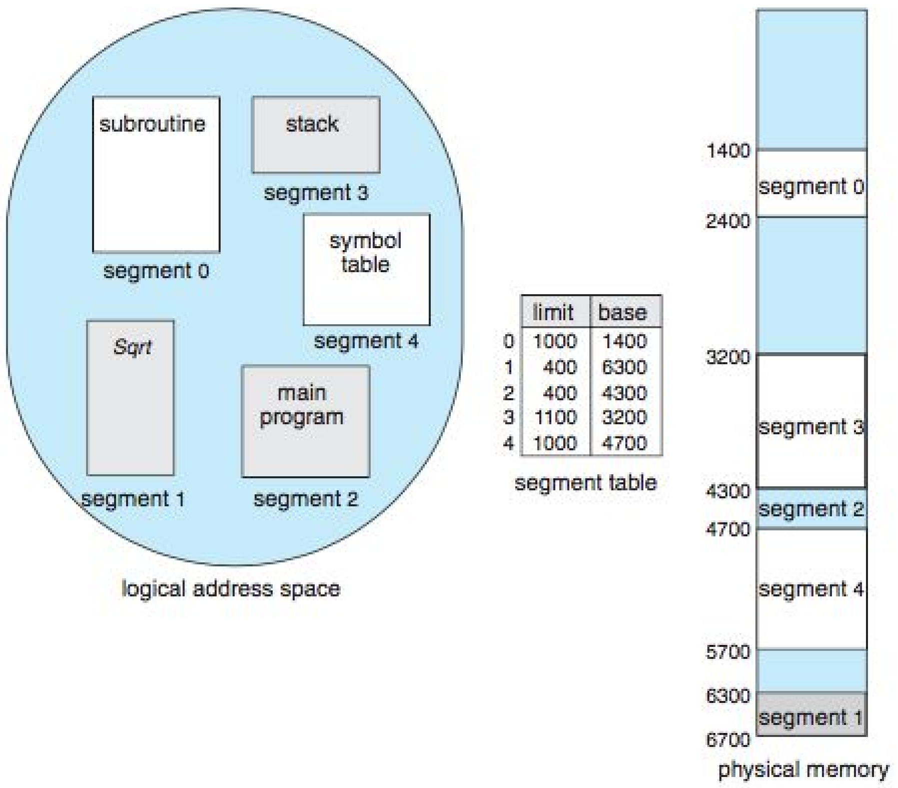


Figure 8.8 Segmentation hardware.

This mapping is effected by a segment table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

As an example, consider the situation shown below We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

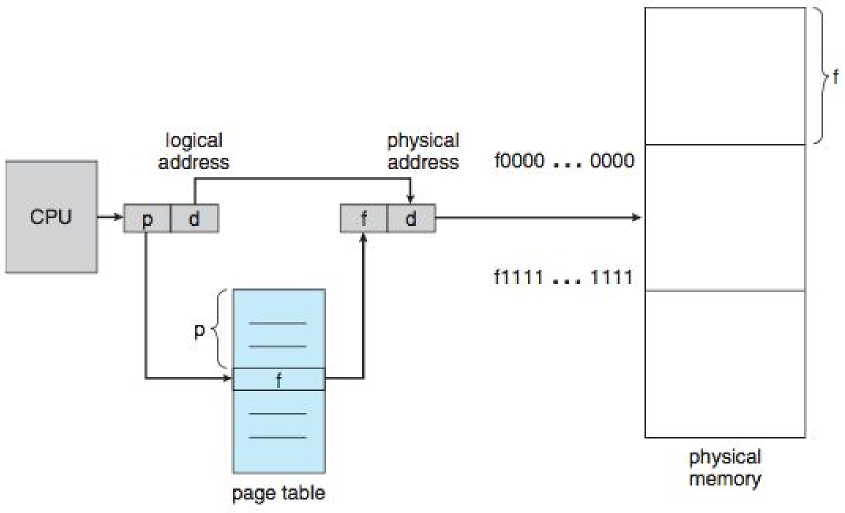
Paging

Paging is another memory-management scheme that offers the advantage of non-contiguous allocation. However, paging avoids external fragmentation and the need compaction, whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.

Basically, as opposed to segmentation in which programs are placed as-is in memory in the most free space, paging splits programs into pages, which are then assigned to locations (called frames) in the memory. **Think of a post person with their post. Their mail are the pages, and the postboxes the frames.**

Paging is also used as a means of Swapping, in which memory blocks are grouped together and placed into the backing store as a group. Physical memory is placed into frames, whilst logical memory is placed into pages. When a program loads, these are put back into memory as full blocks. This solves the issue of varying sizes.

The basic method for implementing paging involves breaking physical memory into **fixed-sized blocks** called **frames** and breaking **logical memory** into blocks of the **same size** called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.



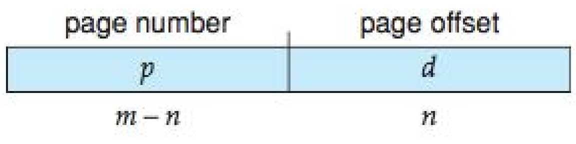
Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table.

The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1GB per page, depending on the computer architecture.

The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m − n bits of a logical address designate the **page number**, and the n low-order bits designate the **page offset**. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

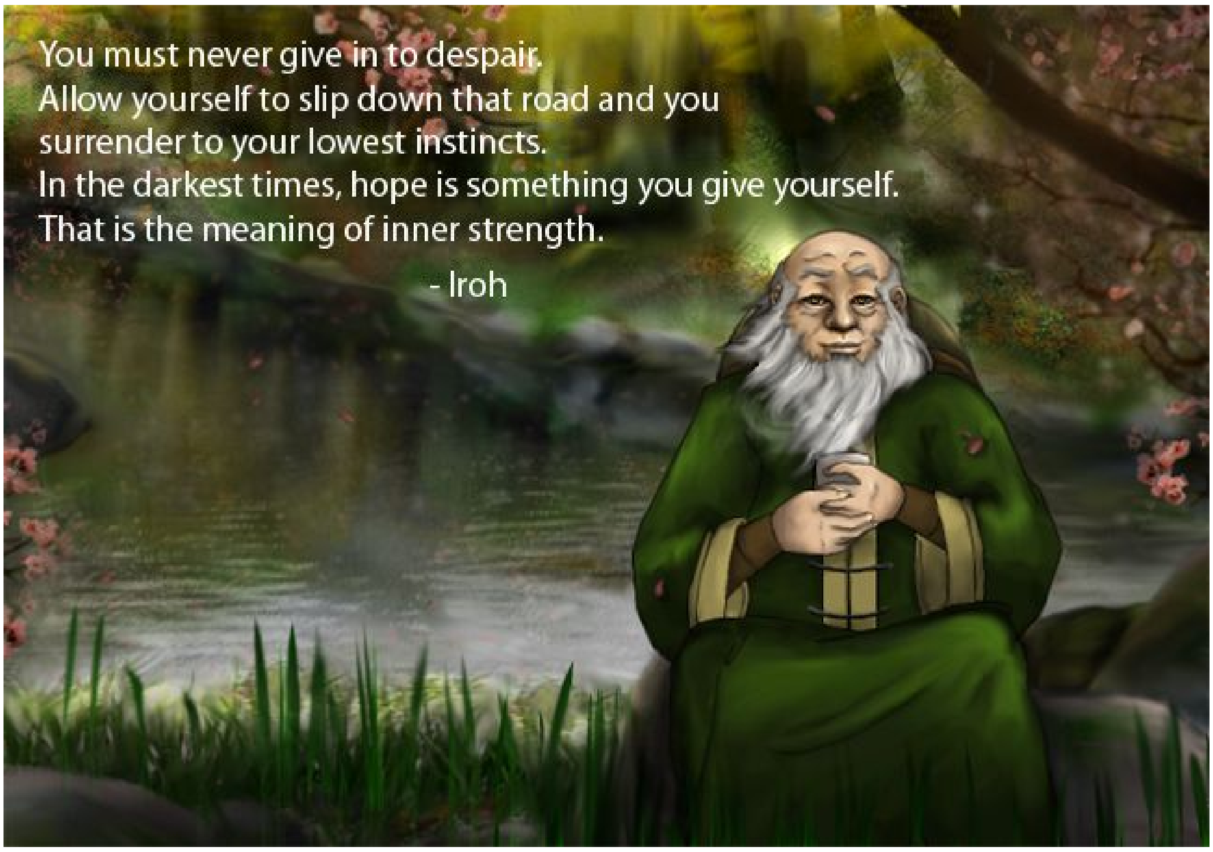
Fragmentation with Paging

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. Basically, no space is wasted between frames in memory.

However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. Thus, if we only need to store 1MB in a 4MB frame, 3MB of internal space is wasted.

For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes.

IT WAS AT THIS POINT WHERE RHYS GAVE UP ON SUMMARISING THE DINOSAUR BOOK. “THERE'S JUST TOO MUCH”, HE SAID, AS HE DRIFTED INTO THE STORM.


You must never give in to despair.
Allow yourself to slip down that road and you surrender to your lowest instincts.
In the darkest times, hope is something you give yourself.
That is the meaning of inner strength.
- Iroh