Design of Digital Circuits
Summary

July 19, 2020

Chapter 1

From Zero to One

1.1 Managing Complexity

Abstraction: The levels of abstraction for an electronic computer:

- **physics:** The motion of electrons
- **electronic devices:** Transistors which have connection points (terminals) and can be modeled by the relationship between voltage and current as measured at each terminal.
- **analog circuits:** Devices which are assembled to create components such as amplifiers. They input and output a continuous range of voltages
- **Digital circuits:** e.g logic gates restrict the voltages to descrete ranges which we use to indicate 0 and 1.
- **Microarchitecture:** Links the logic and architecture levels of abstraction. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture-
- **Architecture:** Describes the computer from the programmers perspective. A particular architecture can be implemented by one of many different microarchitectures.
- **Operating system:** Handles low-level details such as accessing a hard drive or managing memory.
- **Application software:** Uses the facilities provided by the operating system to solve a problem for the user.

Discipline: the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction.

The Three- Y's:

- **Hierarchy:** involves dividing a system into modules then further subdividing each of these modules until the pieces are easy to understand
- **Modularity:** states that the modules have well-defines functions and interfaces so that they connect together easily without unanticipated side effects
- **Regularity:** seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed

1.2 Digital Abstraction:

Discrete-valued variables: Variables with a finite number of distinct values. Most electronic computers use a binary representation in which a high voltage indicates a 1 and a low voltage indicates a 0. The amount of information D in a descrete valued variable with N distinct states is measured in units of bits as:

$$D = log_2 N \text{ bits}$$

hence a binary variable conveys $log_2 2 = 1$ bit of information

1.3 Number Systems

Binaray Numbers: Bits represent one of two values 0 or 1 and are joined together to form binary numbers. Each column of a binary number has twice the weight of the previous column, hence they are base 2 (the base is denoted as subscript e.g 10110_2).

Hexadecimal Numbers: groups of four bits i.e base 16. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Bytes,Nibbles,etc: A group of eight bits is called a byte. The size of objects stored in computer memories is measured in bytes rather than bits. A group of four bits, is called a nibble i.e one hexadecimal digit stores one nibble. Microprocessors handle data in chunks called words which the size of depends on the architecture of the microprocessor (64 bit processors indicate that they operate on 64-bit words)

Most/Least significant bit (lsb/msb): Within a group of bits in the 1's column is called the lsb and the bit at the other end is called the msb

Estimating Powers of Two:

- $2^{10} \approx 10^3$
- $2^{20} \approx 10^6$
- $2^{30} \approx 10^9$

Binary Addition
:

Figure 1.8 Addition examples showing carries: (a) decimal (b) binary

Signed Binary Numbers: The two most widely employed systems to represent signed binary numbers are:

- Sign/magnitude: The msb is used as the sign and the remaining N-1 bits is the absolute value. 0 indicates positive and 1 indicates negative. Ordinary binary addition does not work for sign/magnitude numbers. (0 has two representations)
- Two's complement: Identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. Zero has a single representation and ordinary addition works. The sing of a two's complement number is reversed in a process called taking the two's complement. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. When addint N-bit numbers, the carry out of the Nth bit is discarded. Subtraction is performed by taking the two's complement of the second number, then adding. The range of an N-bit twos complement number spans $[-2^{N-1}, 2^{N-1} - 1]$

1.4 Logic Gates

Logic Gates: are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input and the output. Inputs are drawn on the left or top and recieve a letter near the beginning of the alphabet and the outputs on the right or bottom and recieve the letter Y.

Truth table: Lists the inputs on the left and the corresponding output on the right. One row is designated for each possible combination of inputs.

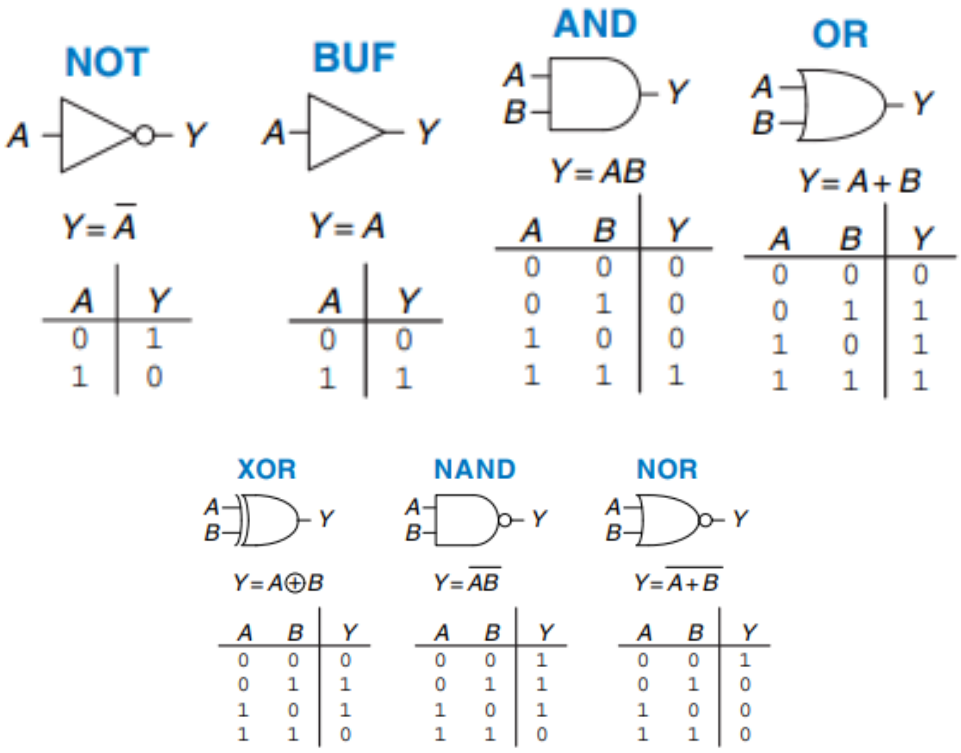


Figure 1.1: Common Gates

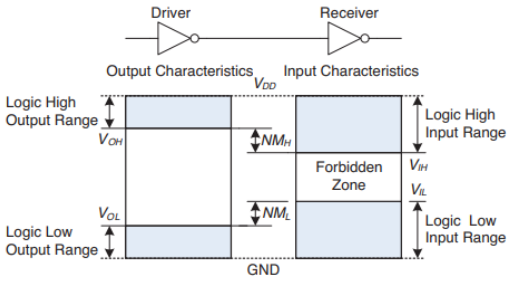
Bubble: circle on the output of a logic gate indicating an inversion.

N-input XOR: Produces true when an odd number of inputs are TRUE.

1.5 Beneath the digital Abstraction

Supply Voltage: The lowest voltage in a system is called ground or GND (0V). The highest voltage comes from the power supply and is usually called V_{DD}.

Logic Levels and Noise Margins: used to map continuous variables onto discrete binary variables.



The driver produces a LOW(0) output in the range of 0 to V_{OL} or a HIGH(1) in the range of V_{OH} to V_{DD}. If the reciever gets an input in the range of 0 to V_{IL} it will consider the input to be LOW, if it gets an input in the range of V_{IH} to V_{DD} it will consider the input to be HIGH. If the receivers input falls into the forbidden zone the behaviour of the gate is unpredictable. $V_{OH}, V_{OL}, V_{IH}, V_{IL}$ are called the output and input high and low logic levels. The noise margin is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input.

$$NM_L = V_{IL} - V_{OL}$$
$$NM_H = V_{OH} - V_{IH}$$

DC transfer characteristics: of a gate describes the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. A reasonable place to choose logic levels is where the slop of the transfer characteristics $\frac{dV(Y)}{dV(A)} = -1$ These two points are called **unity gain points**. This usually maximizes the noise margins.

Static Discipline: Given logically valid inputs, every circuit element will produce logically valid outputs. The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Gates are grouped into logic families such that all gates in a logic family obey the static discipline when used with other gates in the family.

1.6 CMOS Transistors

Transistors: Electrically controlled switches that turn on or off when a voltage or current is applied to a control terminal. There are two main type of transistors:

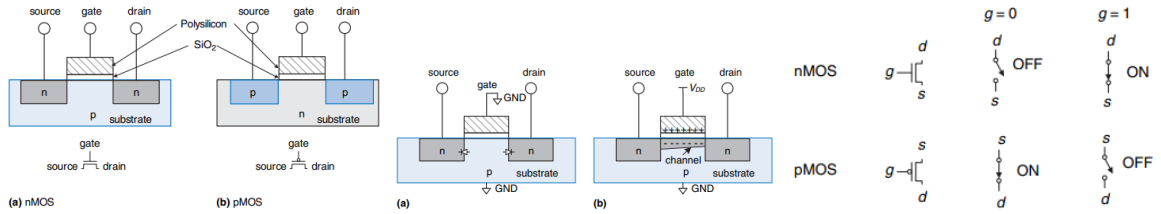
- bipolar junction transistors
- metal-oxide-semiconductor field effect transistors (MOSFETs or MOS transistors)

Semiconductors: MOS transistors are built from silicon. The conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants(impurities in the silicon lattice). Adding a dopant with more electrons than the silicon can bind to creates a negative charge which can move around the lattice and is called an n-type dopant. Adding a dopant with less electrons creates a positive charge and hence called a p-type dopant.

Diodes: Junction between p-type and n-type silicon. The p-type region is called the anode and the n-type region called the cathode. When voltage on the anode rises above the voltage on the cathode, the diode is forward biased, and current flows through the diode from the anode to the cathode otherwise it is reverse biased and no current flows.

Capacitors: Consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric charge and the other conductor accumulates the opposite charge -Q. The capacitance C of the capacitor is the ratio of charge to voltage $C = \frac{Q}{V}$. More capacitance means that a circuit will be slower and require more energy to operate(charging and discharging takes time).

nMOS and pMOS Transistors: n-type transistors called nMOS, have regions of n-type dopants adjacent to the gate and built on a p-type semiconductor substrate. pMOS transistors are the opposite. Chips which have both nMOS and pMOS transistors are called complementary MOS or CMOS.



Chapter 2

Combinational Logic Design

2.1 Basic Definitions:

functional specification: Describes the raltionship between inputs and outputs

timing specification: Describes the delay between inputs changing and outputs responding

combinational circuit: The output depends only on the current values of the inputs. (e.g a logic gate) A combinational circuit is memoryless.

sequential circuit: The output depends on both current and previous values of the inputs. Sequential circuits have memory.

Bus: A bundle of multiple signals. A single line with a slash through it and a number next to it indicates a bus with the number specifying the number of signals

Combinational composition: A circuit is combinational if:

- Every circuit element itself is combinational
- Every node(wire) of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

2.2 Boolean Equations

complement: The complement of a variable/literal A is its inverse $\overline{A}$

product/implicant: The AND of one or more literals

minterm: A product involving all of the inputs to the function

sum: The OR of one or more literals

maxterm: A sum involving all the inputs to the function

order of operations: NOT has the highest precedence followed by AND then OR

Sum of Products Form: Given a truth table we can create a boolean equation by summing each of the minterms for which the outpu Y is TRUE.

Product of Sums Form: We can write a boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE.

2.3 Boolean Algebra

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B=0$ if $B\neq 1$	A1′	$B=1$ if $B\neq 0$	Binary field
A2	$\overline{0}=1$	A2′	$\overline{1}=0$	NOT
A3	$0\bullet 0=0$	A3′	$1+1=1$	AND/OR
A4	$1\bullet 1=1$	A4′	$0+0=0$	AND/OR
A5	$0\bullet 1=1\bullet 0=0$	A5′	$1+0=0+1=1$	AND/OR

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B\bullet 1=B$	T1′	$B+0=B$	Identity
T2	$B\bullet 0=0$	T2′	$B+1=1$	Null Element
T3	$B\bullet B=B$	T3′	$B+B=B$	Idempotency
T4			$\overline{\overline{B}}=B$	Involution
T5	$B\bullet \overline{B}=0$	T5′	$B+\overline{B}=1$	Complements

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B\bullet C=C\bullet B$	T6′	$B+C=C+B$	Commutativity
T7	$(B\bullet C)\bullet D=B\bullet(C\bullet D)$	T7′	$(B+C)+D=B+(C+D)$	Associativity
T8	$(B\bullet C)+(B\bullet D)=B\bullet(C+D)$	T8′	$(B+C)\bullet(B+D)=B+(C\bullet D)$	Distributivity
T9	$B\bullet(B+C)=B$	T9′	$B+(B\bullet C)=B$	Covering
T10	$(B\bullet C)+(B\bullet\overline{C})=B$	T10′	$(B+C)\bullet(B+\overline{C})=B$	Combining
T11	$(B\bullet C)+(\overline{B}\bullet D)+(C\bullet D)$ $=B\bullet C+\overline{B}\bullet D$	T11′	$(B+C)\bullet(\overline{B}+D)\bullet(C+D)$ $=(B+C)\bullet(\overline{B}+D)$	Consensus
T12	$\overline{B_0\bullet B_1\bullet B_2\ldots}$ $=(\overline{B}_0+\overline{B}_1+\overline{B}_2\ldots)$	T12′	$\overline{B_0+B_1+B_2\ldots}$ $=(\overline{B}_0\bullet\overline{B}_1\bullet\overline{B}_2\ldots)$	De Morgan's Theorem

Rules for bubble pushing:

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs
- Pushing bubbles on all gate inputs forward toward the output puts a bubble on the output
- Begin at the output of the circuit and work toward the inputs
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output instead of the complement of the output
- Working backward,draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble draw the preceding gate without an output bubble

2.4 From Logic to Gates

schematic: A diagram of a digital circuit showing the elements and the wires that connect them together. The guidlines for drawing schematics:

- Inputs are on the left (or top) side of a schematic
- Outputs are on the right (or bottom) side of a schematic
- Whenever possible, gates should flow from left to right
- Straight wires are better to use than wires with multiple corners
- Wires always connect as a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing without a dot make no connection

Dont Cares: The symbol X is used to describe inputs that the output doesn't care about. When creating a boolean equation from a truth table with dont cares we can use the sum of products and ignore inputs with X's.

2.5 Multilevel Combinational Logic:

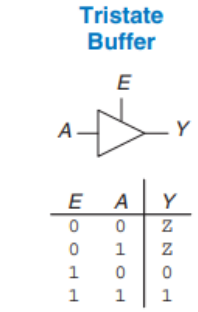
two-level logic: Logic in sum-of-products because it consists of literals connected to a level of AND gates connected to a level of OR gates. Circuits with more than two levels of logic may use less hardware.

2.6 X's and Z's

Illegal Value: X The symbol X indicates that the circuit node has an unkown or illegal value. This happens if it is being driven to both 0 and 1 at the same time. This situation is called **contention** and is considered an error and must be avoided. The voltage at contention is usually in the forbidden zone. In a truth table X is defined as "dont care" i.e it is unimportant if it is a 0 or a 1.

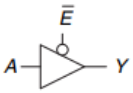
Floating Value: Z This symbol indicates that a node is being drigven neither HIGH nor LOW. A floating node does not always mean there is an error in the circuit, so long as some other circuit element drives the node to a valid logic level when its value is relevant

Tristate Buffer: Commonly used on busses that connect multiple chips. Only one chip at a time is allowed to assert its enable signal to drive a value onto the busses.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

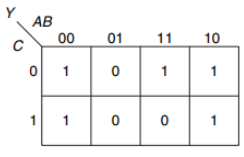
Figure 2.41 Tristate buffer with active low enable

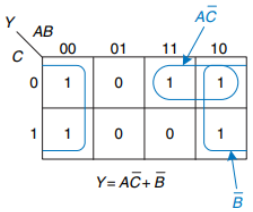
2.7 Karnaugh Maps

Karnaugh maps (K-maps) A graphical method for simplifying Boolean equations. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row (each square represents a single minterm). The combinations of the row are in gray code (00,01,11,10) which ensures entries differ only in a single variable. The K-map wraps around i.e squares on the far right are effectively adjacent to the squares on the far left.

Logic Minimization with K-Maps: We can minimize logic by circling all the rectangular blocks of 1's in the map, using the fewest number of circles. Each circle should be as large as possible. Then read off the implicants that were circled. Rules for finding a minimized equation from a K-map are as follows:

- use the fewest circles necessary to cover all the 1's
- all the squares in each circle must contain 1's
- Each circle must span a rectangular block that is a power of 2
- each circle should be as large as possible
- a circle may wrap around the edges of the K-map
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used

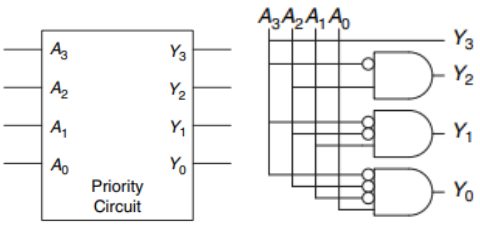




Dont Cares: Dont cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never happen. Such outputs can be treated as either 0's or 1's at the designers discretion. In a K-map they can be circled if they help cover the 1's with fewer or larger circles.

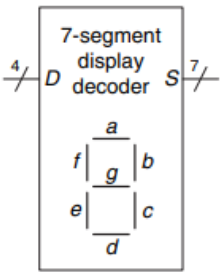
2.8 Combinational Building Blocks:

Priority Circuit: Output with the highest priority signal becomes 1 the rest 0.



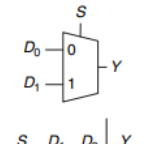
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Seven Segment Display Decoder: Takes a 4-bit data input $D_{3:0}$ and produces seven outputs to control light emitting diodes to display a digit from 0 to 9.



$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

Multiplexers: Chooses an outpu from several possible inputs based on the value of a select signal. (Multiplexer is also called a mux) An N:1 multiplexer needs $log_2 N$ select lines. A 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

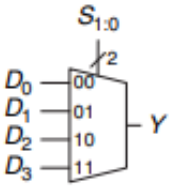
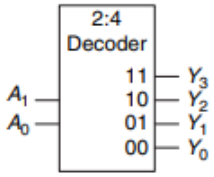


Figure 2.57 4:1 multiplexer

Decoder: A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. An N-input function with M 1's in the truth table can be built with an $N : 2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

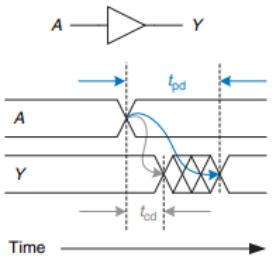
2.9 Timing

delay: The time it takes the output to change in response to an input change. Delay is measured from the 50% point of the input signal to the 50% point of the output signal

rising/falling edge: The transition from LOW to HIGH (rising) and HIGH to LOW (falling).

propagation delay: denoted t_{pd} is the maximum time from when an input changes until the output or outputs reach their final value. It is the sum of the propagation delays through each element on the critical path.

contamination delay: denoted t_{cd} is the minimum time from when an input changes until any output starts to change its value. The sum of contamination delays through each element on the short path.



Critical path: The longest and slowest path in a circuit. The path is critical because it limits the speed at which the circuit operates.

Short path: The shortest and therefore the fastest path through the circuit.

Glitches\Hazards : A single input transition which causes multiple output transitions. In a K-map the transition across the boundry of two prime implicants(bubbles) indicates a possible glitch. To fix this we add another circle that covers that prime implicant boundary

Chapter 3

Sequential Logic Design

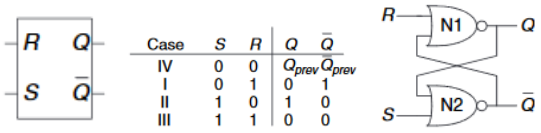
State: The state of a digital sequential circuit is a set of bits called state variables that contain all the information about the past necessary to explain the future behavior of the circuit. An element with N stable states conveys $log_2 N$ bits fo information (a bistable element stores on bit)

3.1 Latches and Flip-Flops

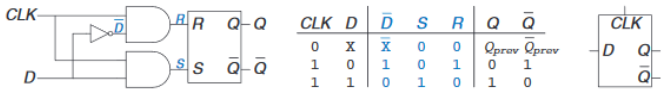
bistable: An element with two stable states

cross-coupled: The input of one element is the output of the other element and vice versa

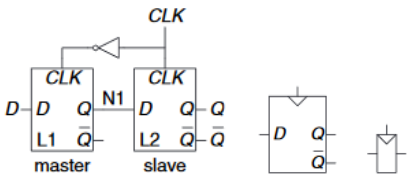
SR-Latch: Consists of two cross-coupled NOR gates. The latch has two inputs, S (set) and R (reset). Setting a bit means to make it TRUE. To reset a bit means to make it FALSE.



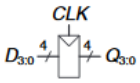
D Latch: Consists of two inputs. The data input D, controls what the next state should be. The clock input CLK controls when the state should change. When CLK = 1 the latch is transparent the data at D flows through to Q as if the latch were just a buffer. When CLK = 0, the latch is opaque, it blocks new data from flowing through to Q, and Q retains the old value



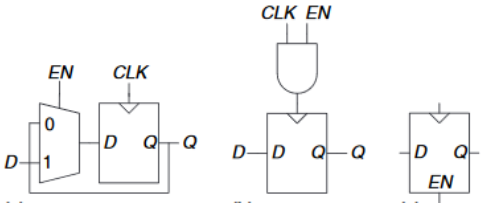
D Flip-Flop: Is built from two back to back D latches controlled by complementary clocks. The first latch L1 is called the master. The second latch L2, is called the slave. The node between them is named N1. When the $\overline{Q}$ is not needed the condensed symbol is used. The D flip flop copies D to Q on the rising edge of the clock, and remembers its state at all other times. Triangle in the symbols denotes an edge-triggered clock input.



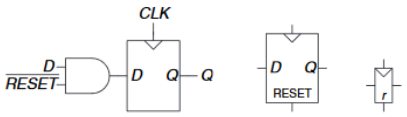
Register: An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.



Enabled Flip-Flop: An enabled flip-flop adds another input EN to determine whether data is loaded on the clock edge. When EN is TRUE the enabled flip-flop behabes like an ordinary D flip-flop. When EN is FALSE the enabled flip-flop ignores the clock and retains its state. Enabled flip flops are useful when we wish to load a new value into a flip flop only some of the time, rather than on every clock edge.



Resettable Flip-flop: A resettable flip flop adds another input called RESET. When RESET is FALSE it behaves like an ordinary D flip-flop. When RESET is TRUE the resettable flip-flop ignores D and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e 0) into all the flip flops in a system when we first turn it on. They can be synchronously (reset themselves only on the rising edge of CLK) or asynchronously resettable (reset as soon as RESET becomes TRUE, independent of CLK). Active low signal means that the reset signal performs its function when it is 0, not 1.



3.2 Synchronous Logic Design

cyclic paths: Outputs are fed directly back to inputs. These circuits are sequential (Combinational logic has no cyclic paths and no races)

synchronized: The state of a system only changes at clock edge

sequential circuit: A circuit with a finite set of discrete states $\{S_0, S_1, \ldots, S_{k-1}\}$

synchronous sequential circuit: A sequential circuit which has a clock input whose rising edge indicate a sequenc of times at which state transitions occur. The timing specification consists of an upper bound t_{pcq} and a lower bound t_{ccq} on the time from the rising edge of the clock until the output changes, aswell as setup and hold times t_{setup}, t_{hold} which indicate when the inputs must be stable relative to the rising edge of the clock. A circuit is a synchronous sequential circuit if:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

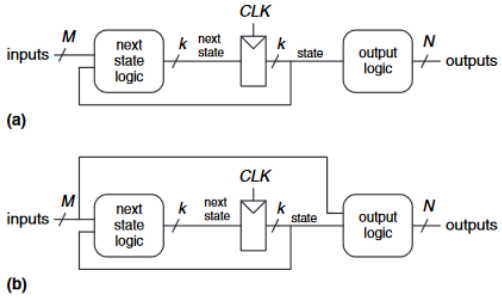
Sequential circuits that are not synchronous are called **asynchronous**. Current state variable is denoted S and the next state variable S'.

3.3 Finite State Machines(FMS)

FSM's: A circuit with k registers can be in one of 2^k unique states. An FSM has M inputs, N outputs and k bits of state. It also recieves a clock and optionally a reset signal. An FSM consists of two blocks of combinational logic, next state logic and output logic and a register that stores the state. On each clock edge the FSM advances to the next state, which was computed based on the current state and inputs.

Moore Machines: FSM where the outputs depend only on the current state of the machine. (a)

Mealy Machines: FSM where the output depends on both the current state and the current inputs (b)



State Transition diagram: Indicates all the possible states of a system and the transitions between these states. Circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock and is not shown in the diagram because a clock is always present in a synchronous sequential circuit. The arcs are labeled with input indicating how it reaches the next state. The value that the outputs have while in a particular state are indicated in the state (for Moore machines). For Mealy machines, the outputs are labeled on the arcs instead of in the circles.

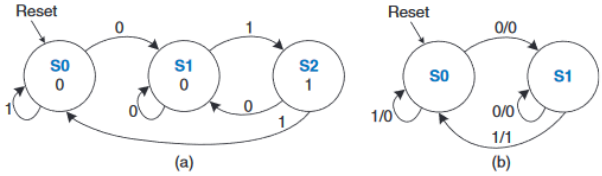
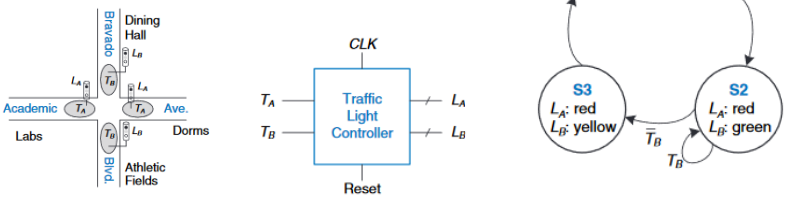


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

State Transition Table: Indicates for each state tand input what the next state should be. The table uses dont care symbols (X) whenever the next state does not depend on a particular input. Reset is omitted from the table. The states and outputs must be assigned binary encodings

Output Table: Indicates for each state what the output should be in that state

Example: Traffic Light: Given inputs T_A, T_B (sensors detecting when people are present) and outputs L_A, L_B (Traffic lights) create the State Machine circuit.



(a) State Transition diagram

Table 3.1 State transition table

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Table 3.4 State transition table with binary encodings

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 Output table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

From the state transition table and output table we can get the boolean equations needed for the next states and outputs.

- $S'_1 = S_1 \oplus S_0$
- $S'_0 = \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B$
- $L_{A1} = S_1$
- $L_{A0} = \overline{S}_1 S_0$
- $L_{B1} = \overline{S}_1$
- $L_{B0} = S_1 S_0$

Using these formulas we can build the next state logic and the output logic and hence the complete circuit.

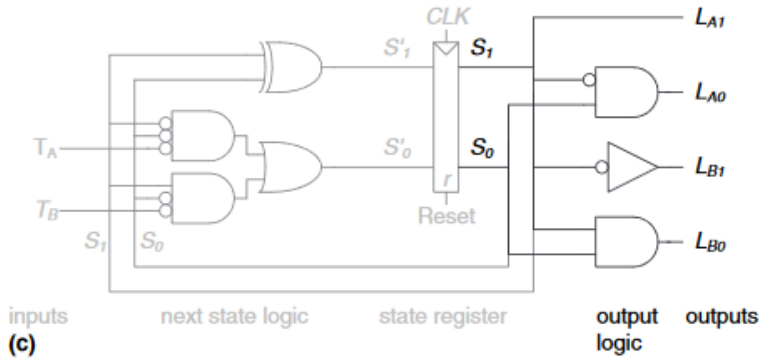


Figure 3.26 State machine circuit for traffic light controller

Binary encoding: Each state is represented as a binary number. A system with K states only need $log_2 K$ bits of state

One-hot encoding: A seperate bit of state is used for each state. Only one bit is "hot" (TRUE) at any time. This encoding requires more flip flops than binary encoding, however the next-state and output logic is often simpler so fewer gates are required. The best encoding choice depends on the specific FSM.

Deriving an FSM from a schematic: Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design:

- Examine circuit, stating inputs,outputs, and state bits
- Write next state and output equations
- Create next state and output tables
- Reduce the next state table to eliminate unreachable states
- Assign each valid state bit combination a name
- Rewrite next state and output tables with state names
- Draw state transition diagram
- State in words, what the FSM does

3.4 Timing of sequential Logic

aperture time: The total time for which the input must remain stable, consisting of the time t_setup before the rising edge of the clock and t_hold after the rising edge. Dynamic discipline states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge, hence we guarantee that the flip flops sample signals while they are not changing

clock to Q contamination delay: denoted t_ccq time after clock rises for which the outputs may start to change

clock to Q propagation delay: denoted t_pcq time after clock rises for which the outputs must settle to final value

clock period/cycle time: denoted T_c is the time between rising edges of a repetitive clock signal

clock frequency: $f_c = \frac{1}{T_c}$. Increasing the clock frequency increases the work that a digital system can accomplish per unit of time. Frequency is measured in units of Hertz (Hz)

setup time constraint/max-delay constraint: $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$. Ideally the entire cycle time T_c would be available for useful computation in the combinational logic t_{pd} however the **sequencing overhead** $(t_{pcq} + t_{setup})$ of the flip flop cuts into this time.

hold time constraint/min-delay constraint: $t_{cd} \geq t_{hold} - t_{ccq}$. This constraint limits the minimum delay through combinational logic. (i.e if two flip flops are connected back to back then $t_{cd} = 0$) In general adding buffers can usually (not always) solve hold time problems without slowing the critical path.

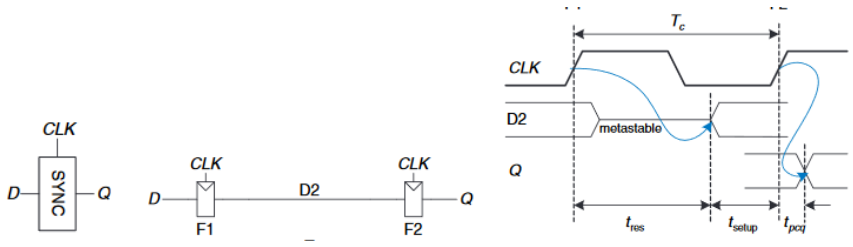
Clock skew: The variation at which the clock signal reaches all the registers. The setup and hold time constraints become:

- $T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$
- $t_{cd} \geq t_{hold} + t_{skew} - t_{ccq}$

Clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic.

Metastable State: When a flip-flop samples an input that is changing during its aperture the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. Eventually it will resolve the output to a stable state(either 0 or 1). The resolution time (time required to reach the stable state) is unbounded. Every bistable device has a metastable state between the two stable states.

Synchronizer: A device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time. The output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.



A synchronizer fails if Q becomes metastable. This may happen if D2 has not resolved to a valid level by the time it must setup at F2.

3.5 Parallelism

Token: A group of inputs that are processed to produce a group of outputs.

Latency: Time required for one token to pass through the system from start to end.

Throughput: The number of tokens that can be produced per unit of time. Throughput can be improved by processing several tokens at the same time.

Spatial parallelism: Multiple copies of the hardware are provided so that multiple tasks can be done at the same time.

Temporal parallelism/Pipelining: A task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap.

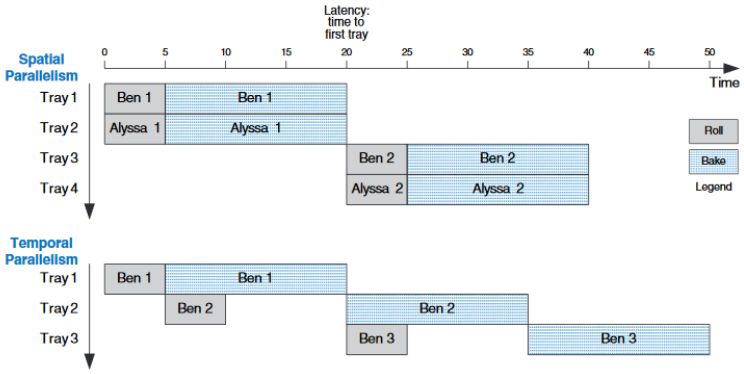


Figure 3.57 Spatial and temporal parallelism in the cookie kitchen

Chapter 4

Hardware Description Languages

4.1 Basics

Module: A block of hardware with inputs and outputs (e.g AND gate, multiplexer, priority circuit etc.)

Behavioral models: Describe what a module does.

Structural models: Describe how a module is built from simpler pieces.

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. ~ indicates NOT, & indicates AND, and | indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

4.2 Combinational Logic

Inverters and Logic Gates: Bitwise operators act on single bit signals or on multi-bit busses.

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

`~`, `^`, and `|` are examples of SystemVerilog *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a & b`, or `~(a | b)`, is called an *expression*. A complete command such as `assign y4 = ~(a & b);` is called a *statement*.

`assign out = in1 op in2;` is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the `=` in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

Comments and Name giving: Be consistent in the use of capitalization and underscores in signal and module names. Module and signal names must not begin with a digit.

Reduction Operators: Reduction operators imply a multiple input gate acting on a single bus.

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line.

SystemVerilog is case-sensitive. `y1` and `Y1` are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

Conditional Assignment: Selects the output from among alternatives based on an input called the condition.

SystemVerilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y = d1`. If `s` is 0, then `y = d0`.

`?:` is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

SystemVerilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

internal variables: They are neither inputs nor outputs but are used only internal to the module. (local variables in programming languages). The order in which statements are written does not matter HDL assignment statements are evaluated any time the inputs, signals on the right hand side change their value.

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Precedence and Numbers: Underscores in numbers are ignored and can be helpful in breaking long numbers into mre readable chunks

SystemVerilog

Table 4.1 SystemVerilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Lowest	?:	Conditional

The operator precedence for SystemVerilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g | p & cin;
```

SystemVerilog

The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal. If the base is omitted, it defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Table 4.3 SystemVerilog numbers

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

Z's and X's: HDL's use z to indicate a floating value and x to indicate an invalid logic level. If a gate receives a floating input, it may produce an x output when it cant determine the correct output value. Similarly if it recieves an illegal or uninitialized input it may produce an x output.

SystemVerilog

SystemVerilog signal values are `0`, `1`, `z`, and `x`. SystemVerilog constants starting with `z` or `x` are padded with leading `z`'s or `x`'s (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns `0` because the output of an AND gate is always `0` if either input is `0`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in SystemVerilog.

Table 4.5 SystemVerilog AND gate truth table with `z` and `x`

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that y is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also note that a `tri` output from a module can be used as a `logic` input to another module. Section 4.7 further discusses nets with multiple drivers.

Bit Swizzling: When operating on a subset of a bus or concatenating signals to form busses. In the example below y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The `{}` operator is used to concatenate busses. `{3{d[0]}}` indicates three copies of `d[0]`.

Don't confuse the 3-bit binary constant `3'b101` with a bus named `b`. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of `y`.

If `y` were wider than 9 bits, zeros would be placed in the most significant bits.

HDL Example 4.13 LOGIC GATES WITH DELAYS

SystemVerilog

```
`timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `'timescale unit/precision`. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as non-blocking (`<=`) and blocking (`=`) assignments, which will be discussed in Section 4.5.4.

4.3 Structural Modeling

structural modeling: Describes a module in terms of how it is composed of simpler modules. Multiple instances of the same module are distinguished by distinct names.

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the SystemVerilog code — see HDL Example 4.5, 4.15, or 4.34.

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

4.4 Sequential Logic:

Registers and Resettable Registers: Generally it is good practice to use resttable registers so that on powerup you can put your system in a known state

SystemVerilog

```
module flop(input  logic      clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

In general, a SystemVerilog `always` statement is written in the form

```
always @(sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

`<=` is called a *nonblocking assignment*. Think of it as a regular `=` sign for now; we'll return to the more subtle points in Section 4.5.4. Note that `<=` is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

SystemVerilog

```
module flopr(input  logic      clk,
             input  logic      reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic      clk,
             input  logic      reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

enabled registers & multiple registers: Enable registers respont to the clock only when the enable is asserted. Always statement can be used to describe multiple pieces of hardware.

SystemVerilog

```
module flopenr(input  logic      clk,
               input  logic      reset,
               input  logic      en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to `{}` in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

D Latch: Avoid using D-latches and use edge triggered flip flops instead

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` is equivalent to `always @(clk, d)` and is the preferred idiom for describing a latch in SystemVerilog. It evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`, so this code describes a positive level sensitive latch. Otherwise, `q` keeps its old value. SystemVerilog can generate a warning if the `always_latch` block doesn't imply a latch.

4.5 4.5 More Combinational Logic

SystemVerilog always statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed. However, always statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL's support blocking and nonblocking assignments in an always statement. A group of blocking assignments are evaluated in the order in which they appear in the code. Nonblocking assignments are evaluated concurrently. All the statements are evaluated before any of the signals on the left hand sides are updated

Inverter and Fulladder implementation with always statement

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y=~a;
endmodule
```

`always_comb` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = in the `always` statement change. In this case, it is equivalent to `always @(a)`, but is better because it avoids mistakes if signals in the `always` statement are renamed or added. If the code inside the `always` block is not combinational logic, SystemVerilog will report a warning. `always_comb` is equivalent to `always @(*)`, but is preferred in SystemVerilog.

The = in the `always` statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In SystemVerilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p=a ^ b;          // blocking
      g=a & b;          // blocking

      s=p ^ cin;        // blocking
      cout=g | (p & cin);  // blocking
    end
endmodule
```

In this case, `always @(a, b, cin)` would have been equivalent to `always_comb`. However, `always_comb` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

Case Statements: The case statement must appear inside an always statement. The case statement performs different actions depending on the value of its input. It implies combinational logic if all possible input combinations are defined otherwise it implies sequential logic, because the output will keep its old value in the undefined cases.

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                    abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

The `default` statement isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x` or `z`.

If Statements: always statements may also contain if statements. The if statement may be followed by an else statement.It implies combinational logic if all possible input combinations are defined otherwise it implies sequential logic.

Priority Circuit:

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y <= 4'b1000;
    else if (a[2]) y <= 4'b0100;
    else if (a[1]) y <= 4'b0010;
    else if (a[0]) y <= 4'b0001;
    else           y <= 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside of `always` statements.

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y <= 4'b1000;
      4'b01??: y <= 4'b0100;
      4'b001?: y <= 4'b0010;
      4'b0001: y <= 4'b0001;
      default: y <= 4'b0000;
    endcase
endmodule
```

The `casez` statement acts like a `case` statement except that it also recognizes `?` as don't care.

The right image is a priority circuit using dont cares.

Blocking and Nonblocking assignments: Guidelines:

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

4.6 Finite State Machines

Divide By 3 FSM:

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate <= S1;
      S1:      nextstate <= S2;
      S2:      nextstate <= S0;
      default: nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `typedef` statement defines `statetype` to be a two-bit `logic` value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```
typedef enum logic [2:0] {S0=3'b001, S1=3'b010, S2=3'b100}
statetype;
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

The above example uses an asynchronouzs reset to initialize the FSM. Next-state and output logic blocks are combinational.
Pattern Recognizer: Moore (Left) and Mealy (Right)

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate=S0;
          else   nextstate=S1;
      S1: if (a) nextstate=S2;
          else   nextstate=S1;
      S2: if (a) nextstate=S0;
          else   nextstate=S1;
      default:   nextstate=S0;
    endcase

  // output logic
  assign y=(state==S2);
endmodule
```

Note how nonblocking assignments (<=) are used in the state register to describe sequential logic, whereas blocking assignments (=) are used in the next state logic to describe combinational logic.

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate=S0;
          else   nextstate=S1;
      S1: if (a) nextstate=S0;
          else   nextstate=S1;
      default:   nextstate=S0;
    endcase

  // output logic
  assign y=(a & state==S1);
endmodule
```

4.7 Data Types

Prior to SystemVerilog, Verilog used two types: reg and wir. reg might or might not be assaciated with a register. In Verilog if a signal appears on the left hand side of <= or = in an always block, it must be declared as reg, otherwise it should be declared as wire. Hence a reg signal may be the output of a flip-flop, latch or combinational logic, depending on the sensitivity list and statement of an always block. Input and output ports default to the wire type unless their type is explicitly defined as reg. The following is an example of a flip-flop implemented in Verilog.

```
module flop(input           clk,
            input     [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog introduces the logic type, which is a synonym for reg and avoids misleading users about whether it is actually a flip-flop. logic can be used outside always blocks where wire traditionally would have been required i.e nearly all SystemVerilog signals can be logic. The exception is that signals with multiple drivers must be declared as a net. The most common typ of net is called a wire or tri. (convention: wire when a single driver is present but is obsolete in Systemverilog, tri when multiple drivers.) When a tri net is driven to a single value by one or more drivers, it takes on that value. When it is undriven, it floats (z). When it is driven to a different value (0,1,x) by multiple drivers, it is in contention (x).

4.8 Parameterized Modules

HDL's permit variable bit widths using parameterized modules.
Parameterized N-bit 2:1 Mux, N-input AND gate and $N : 2^N$ Decoder:

SystemVerilog

```
module mux2
  #(parameter width=8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y=s ? d1 : d0;
endmodule
```

SystemVerilog allows a `#(parameter ...)` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, in this case called `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#( )` before the instance name, as shown below.

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(...)` in defining and overriding parameters.

SystemVerilog

```
module andN
  #(parameter width=8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0]=a[0];
    for(i=1; i<width; i++i+1) begin: forloop
      assign x[i]=a[i] & x[i-1];
    end
  endgenerate

  assign y=x[width-1];
endmodule
```

The `for` statement loops through $i = 1, 2, \ldots,$ `width-1` to produce many consecutive AND gates. The `begin` in a `generate for` loop must be followed by a `:` and an arbitrary label (`forloop`, in this case).

SystemVerilog

```
module decoder
  #(parameter N=3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y=0;
      y[a]=1;
    end
endmodule
```

`2**N` indicates 2^N.

HDL's provide "generate" statements to produce a variable amount of hardware depending on the value of a parameter. generate supports for loops and if statements to determine how many of what types of hardware to produce.

4.9 Testbenches

Testbench: an HDL module that is used to test another module, called the device under test (DUT). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called test vectors. e.g:

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a=0; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
    a=1; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

SystemVerilog

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a=0; b=0; c=0; #10;
    assert (y === 1) else $error("000 failed.");
    c=1;  #10;
    assert (y === 0) else $error("001 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("010 failed.");
    c=1;  #10;
    assert (y === 0) else $error("011 failed.");
    a=1; b=0; c=0;  #10;
    assert (y === 1) else $error("100 failed.");
    c=1;  #10;
    assert (y === 1) else $error("101 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("110 failed.");
    c=1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

The SystemVerilog `assert` statement checks if a specified condition is true. If not, it executes the `else` statement. The `$error` system task in the `else` statement prints an error message describing the assertion failure. `assert` is ignored during synthesis.

In SystemVerilog, comparison using == or `!=` is effective between signals that do not take on the values of `x` and `z`. Testbenches use the `===` and `!==` operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`.

Writing code for each test vector is tedious. A better approach is to place the test vectors in a seperate file. The testbench reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vectors, and repeats until reaching the end of the test vectors file.

SystemVerilog

```
module testbench3();
  logic         clk, reset;
  logic         a, b, c, y, yexpected;
  logic [31:0]  vectornum, errors;
  logic [3:0]   testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk=1; #5; clk=0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum=0; errors=0;
      reset=1; #27; reset=0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin  // check result
        $display("Error: inputs=%b", {a, b, c});
        $display(" outputs=%b (%b expected)", y, yexpected);
        errors=errors+1;
      end
      vectornum = vectornum+1;
      if (testvectors[vectornum]=== 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$readmemh` is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs (a, b, and c) and the expected output (yexpected) based on the four bits in the current test vector.

The testbench compares the generated output, y, with the expected output, yexpected, and prints an error if they don't match. `%b` and `%d` indicate to print the values in binary and decimal, respectively. `$display` is a system task to print in the simulator window. For example, `$display("%b %b", y, yexpected);` prints the two values, y and yexpected, in binary. `%h` prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the `testvectors` array. `$finish` terminates the simulation.

Note that even though the SystemVerilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

4.10 Other Circuits

- **CPA**

SystemVerilog

```
module adder #(parameter N = 8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_
use IEEE.NUMERIC_STD_UNSIGNED.ALL

entity adder is
   generic(N: integer := 8);
   port(a, b: in  STD_LOGIC_VECTO
        cin:  in  STD_LOGIC;
        s:    out STD_LOGIC_VECTO
        cout: out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR
begin
  result <= ("0" & a) + ("0" & b) +
  s      <= result(N-1 downto 0);
  cout   <= result(N);
end;
```

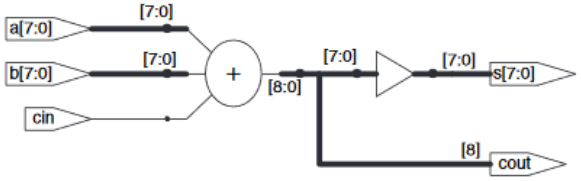


Figure 5.8 Synthesized adder

- **Subtractor:**

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

   assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.
use IEEE.NUMERIC_STD_U

entity subtractor is
  generic(N: integer :-
  port(a, b: in  STD_L
       y:    out STD_L
end;

architecture synth of s
begin
  y <= a - b;
end;
```

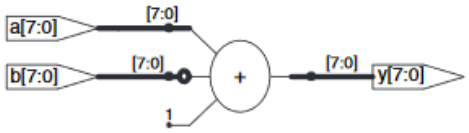


Figure 5.10 Synthesized subtractor

- Comparator:

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LO

entity comparators is
  generic(N: integer := 8);
  port(a, b: in STD_LOGIC_VEC
       eq, neq, lt, lte, gt, g
end;

architecture synth of comparat
begin
  eq  <= '1' when (a = b)  else
  neq <= '1' when (a /= b) else
  lt  <= '1' when (a < b)  else
  lte <= '1' when (a <= b) else
  gt  <= '1' when (a > b)  else
  gte <= '1' when (a >= b) else
end;
```

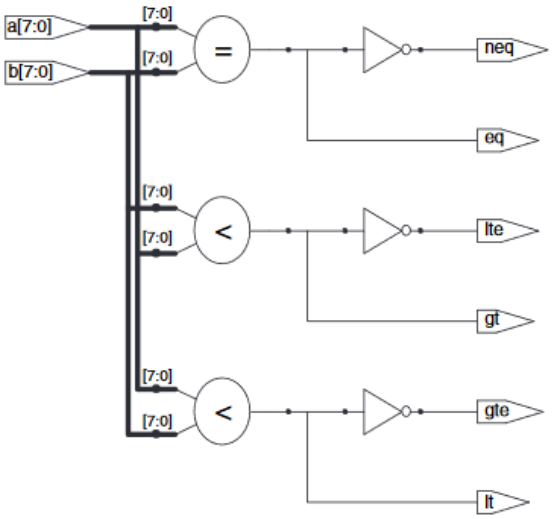


Figure 5.13 Synthesized comparators

- Multiplier:

SystemVerilog

```
module multiplier #(parameter N = 8)
                   (input  logic [N-1:0]   a, b,
                    output logic [2*N-1:0] y);

  assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.
use IEEE.NUMERIC_STD_U

entity multiplier is
  generic(N: integer :
  port(a, b: in  STD_L
       y:    out STD_L
end;

architecture synth of m
begin
  y <= a * b;
end;
```

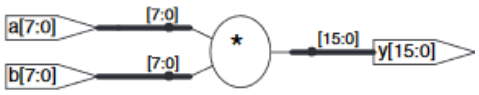


Figure 5.19 Synthesized multiplier

- Counter:

SystemVerilog

```
module counter #(parameter N = 8)
                (input  logic clk,
                 input  logic reset,
                 output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_
use IEEE.NUMERIC_STD_UNSIG

entity counter is
  generic(N: integer := 8);
  port(clk, reset: in  STD_
       q:          out STD_
end;

architecture synth of count
begin
  process(clk, reset) begi
    if reset then
    elsif rising_edge(clk)
    end if;
  end process;
end;
```

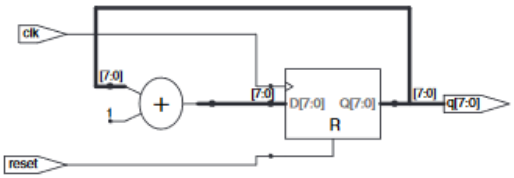


Figure 5.32 Synthesized counter

- Counter:

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic         clk,
                  input  logic         reset, load,
                  input  logic         sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic         sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)       q <= 0;
    else if (load)   q <= d;
    else             q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164

entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECT
       q:          out STD_LOGIC_VECT
       sout:       out STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process(clk, reset) begin
    if reset = '1' then   q <= (OTHERS =
    elsif rising_edge(clk) then
      if load then        q <= d;
      else                q <= q(N-2 dow
      end if;
    end if;
  end process;

  sout <= q(N-1);
end;
```

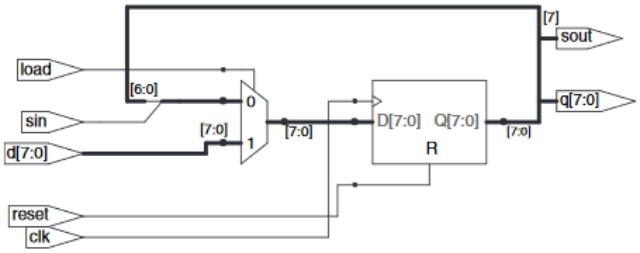


Figure 5.36 Synthesized shiftreg

- RAM:

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGI
use IEEE.NUMERIC_STD_UNSIGNED.

entity ram_array is
  generic(N: integer := 6; M: in
  port(clk,
       we:   in  STD_LOGIC;
       adr:  in  STD_LOGIC_VEC
       din:  in  STD_LOGIC_VEC
       dout: out STD_LOGIC_VEC
end;

architecture synth of ram_array
  type mem_array is array ((2**
       of STD_LOGIC_VECTOR (M-1
  signal mem: mem_array;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we then mem(TO_INTEGE
      end if;
    end if;
  end process;

  dout <= mem(TO_INTEGER(adr))
end;
```

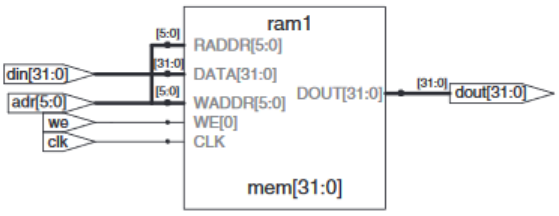


Figure 5.53 Synthesized ram

- ROM:

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout):

  always_comb
    case(adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

Chapter 5

Digital Building Blocks

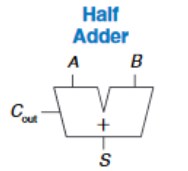
5.1 Arithmetic Circuits

Computers and digital logic perform many arithmetic functions:

- addition
- subtraction
- comparison
- shift
- multiplication
- division

Addition:

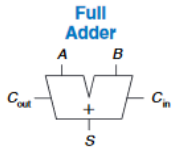
- **Half Adder:** The half adder has two inputs A and B and two outputs S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2 which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half adder can be built from an XOR gate and an AND gate.
- **Full Adder:** Similar to the half adder but accepts a carry in C_{in}



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

Figure 5.1 1-bit half adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Figure 5.3 1-bit full adder

- **Carry Propagate Adder:** An N-bit adder which sums two N-bit inputs A and B, and a carry in C_in to produce an N-bit result S and a carry out C_out. It is called a carry propagate adder because the carry out of one bit propagates into the next bit.
- **Ripple-Carry Adder:** The simplest way to build an N-bit carry propagate adder is to chain together N full adders. This is called a ripple-carry adder. The C_{out} of one stage acts as the C_{in} of the next stage. The ripple carry adder has the disadvantage of being slow when N is large. The delay of the adder t_{ripple}, grows directly with the number of bits:

$$t_{ripple} = Nt_{FA}$$

where t_{FA} is the delay of the full adder

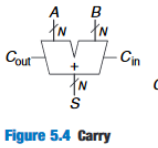


Figure 5.4 Carry propagate adder

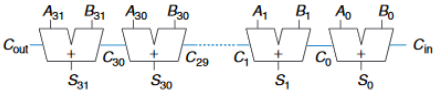


Figure 5.5 32-bit ripple-carry adder

- **Carry-Lookahead Adder:** Another type of carry propagate adder that divides the adder into blocks provides circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to look ahead across the blocks rather than waiting to ripple through all the full adders inside a block. CLA's use generate (G) and propagate (P) signals that describe how a column or block determines the carry out. The i-th column of an adder is said to generate a carry if it produces a carry out independent of the carry in (i.e if A_i and B_i are both 1). Hence G_i , the generate signal for column i, is calculated as $G_i = A_i B_i$. The column is said to propagate a carry if it produces a carry out whenever there is a carry in. The ith column will propagate a carry in C_{i-1} if either A_i or B_i is 1, hence $P_i = A_i + B_i$

$$C_i = A_i B_i + (A_i + B_i)C_{i-1} = G_i + P_i C_{i-1}$$

The generate and propagate definitions extend to multiple-bit blocks:

$$G_{i:j} = G_i + P_i(G_{i-1} + P_{i-1}(... + P_{j+2}(G_{j+1} + P_{j+1}G_j)))$$
$$P_{i:j} = P_i P_{i-1} ... P_{j+1} P_j$$
$$C_i = G_{i:j} + P_{i:j} C_j$$

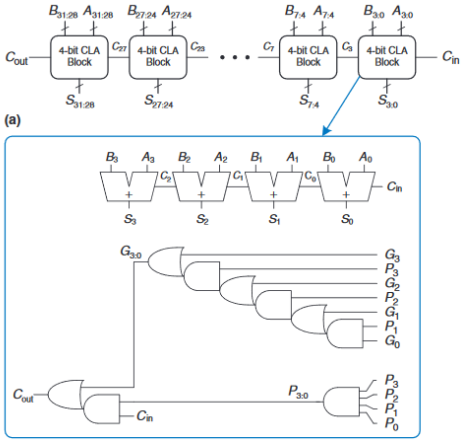
All of the CLA blocks compute the single-bit, generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. This is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally the critical path through the last block contains a short ripple-carry adder, hence an N-bit adder divided into k-bit blocks has a delay:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (\frac{N}{k} - 1)t_{AND\_OR} + kt_{FA}$$

where:

- t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate P_i, G_i
- t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}, G_{i:j}$ for a k-bit block
- t_{AND_OR} is the delay from C_{in} to C_{out} through the final AND/OR logic of teh k-bit CLA block

For $N > 16$ the CLA is much faster than the ripple-carry adder, however the delay still increases linearly with N.



- **Prefix Adder:** Extend the generate and propagate logic of the CLA to perform addition even faster. They first compute G and P for pairs of columns then for blocks of 4,8,16,... until the generate signal for every column is known. The sums are computed from these generate signals. I.e the prefix adder tries to compute the carry in for each column i as quickly as possible, then compute the sum using:

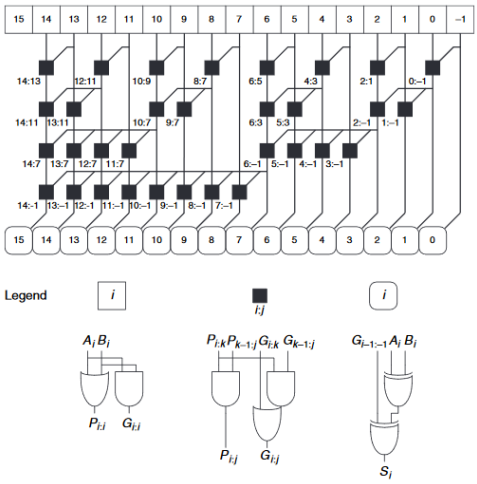
$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

Define column i=-1 to hold C_{in} so $G_{i-1} = C_{in}$ and $P_{-1} = 0$ Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column i-1 if the block spanning columns i-1 through -1 generates a carry. Hence:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

The main challenge is to rapidly compute all the block generate signals $G_{-1:-1}, G_{0:-1}, ..., G_{N-2:-1}$ These signals along with $P_{-1:-1}, P_{0:-1}, ..., P_{N-2:-1}$ are called prefixes. A Prefix adder begins with a precomputation to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $log_2 N$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. It combines these parts to form generate and propagate signals for the entire block spanning bits $i : j$ using the equatios:

$$G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j}$$
$$P_{i:j} = P_{i:k}P_{k-1:j}$$



The Prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. The speedup is significant but comes at the expense of more hardware than a CLA. The network of black cellse is called a prefrix tree. The critical path for an N-bit prefix adder involves the precomputation of P_i and G_i followed by $log_2 N$ stages of black prefix cells to obtain all the prefixes $G_{i-1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i:

$$t_{PA} = t_{pg} + log_2 N(t_{pg\_prefix}) + t_{XOR}$$

(t_{pg_prefix} is the delay of a black prefix cell)

Subtraction: Subtraction is done by flipping the sign of the second number, then adding it. Flipping the sign of a twos complement is done by inverting the bits and adding 1. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{in} = 1$

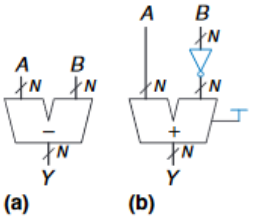
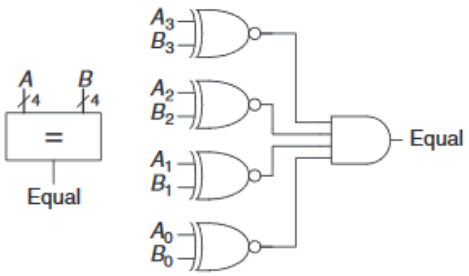


Figure 5.9 Subtractor: (a) symbol, (b) implementation

Comparators: A comparator determines whether two binary numbers are equal or if one is greater or less than the other. It receives two N-bit binary numbers A and B. There are two common types of comparators:

- **Equality comparator:** Produces a single output indicating whether A is equal to B



- **Magnitude comparator:** Produces one or more outputs indicating the relative values of A and B. Magnitude comparison is usually done by computing A-B and looking at the sign (most significant bit) of the result-

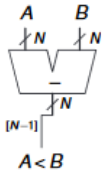
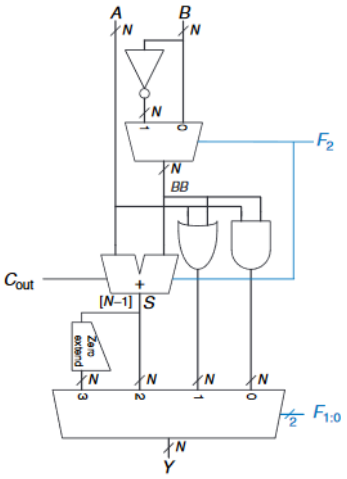


Figure 5.12 ***N*-bit magnitude comparator**

Arithmetic Logic Unit (ALU): An ALU combines a variety of mathematical and logical operations into a single unit. It recieves a control signal F that specifies which function to perform. Control signals will generally be shown in blue to distinguish them from the data. Typifal functions an ALU can perform:

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



The right figure is an implementation of the ALU. The ALU contains an N-bit adder and N two-input AND and OR gates. It also contains inverters and a multiplexer to invert input B when F_2 control signal is asserted. A 4:1 multiplexer chooses the desired function based on the $F_{1:0}$ control signals. More specifically the arithmetic and logical blocks in the ALU operate on A and BB. If $F_{1:0}$ is:

- 00 → ALU computes A AND BB
- 01 → ALU computes A OR BB
- 10 → ALU performs addition or subtraction (note F_2 is a carry in to the adder and if its 1 then ALU computes A $+\overline{B} + 1$ = A-B
- 11 and $F_2 = 1$ → ALU performs SLT (set if less than) operation. When $A < B, Y = 1$ otherwise Y=0. SLT is performed by computing S=A-B. If S is negative i.e the sign bit is set, A is less than B. The zero extend unit produces an N-bit output by concatenating its 1-bit input with 0's in the most significant bits. The sign bit (N-1th bit) of S is the input to the zero extend unit. Some ALUs produce extra outputs called flags that indicate information about the ALU output (e.g overflow,zero)

  Shifters and Rotators: They move bits and multiply or divide by powers of 2. A Shifter shifts a binary number left or right by a specified number of positions. Several common types:

  - **Logical Shifter:** Shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's
  - **Arithmetic shifter:** Same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit. This is useful for multiplying and dividing signed numbers
  - **Rotator:** Rotates numer in circle such that empty spots are filled with bits shifted off the other end.

  An N-bit shifter can be built from N N:1 multiplexers. The input is shifted depending on the value of the $log_2 N$ bit select lines. The operators $<<, >>, >>>$ typically indicate shift left, logical shift right and arithmetic shift right respectively. Depending on the value of the 2-bit shift amount $shamt_{1:0}$ the output Y receives the input A shifted by 0 to 3 bits. (A left shift by N bits corresponds to a multiplication by 2^N, A right shift is a division by 2^N)

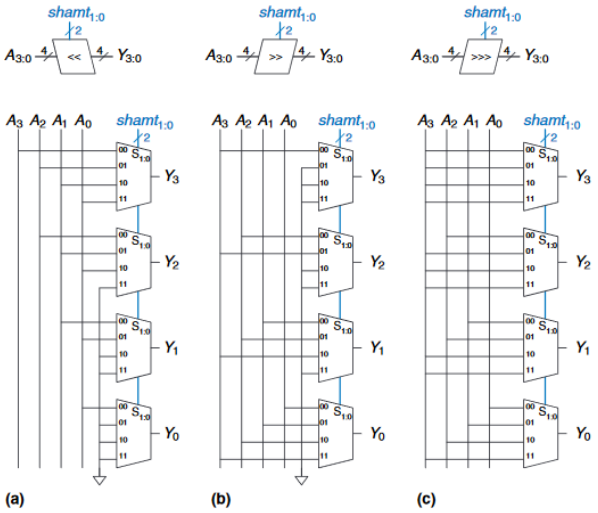
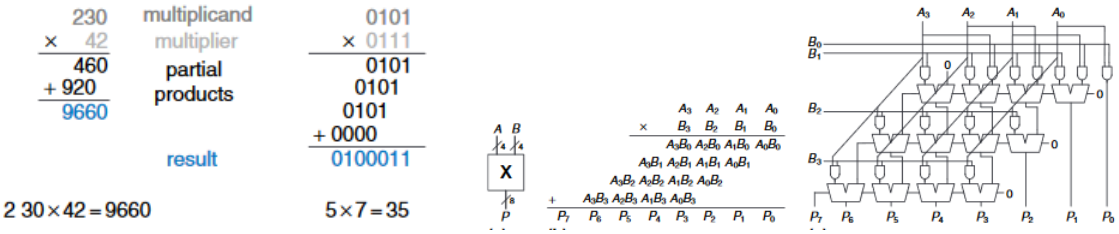


Figure 5.16 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

Multiplication: Partial products are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result. In general an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N - bit result$. Multiplication of 1-bit binary numbers is equivalent to the AND operation, hence AND gates are used to form the partial products.



5.2 Number Systems

Zhid drvzion introduces fixed- and floating point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some bits represent the integer part, and the rest represent the fraction. Floating point numbers are analogous to scientific notation, with a mantissa and an exponent.

Fixed-Point Number Systems: Fixed point notation has an implied binary point between the integter and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. Signed fixed-point numbers can use either twos complement or sign/magnitude notation

(a) 01101100

(b) 0110.1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

Figure 5.21 Fixed-point notation of 6.75 with four integer bits and four fraction bits

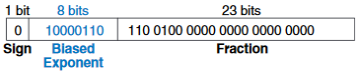
(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.22 Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement

Floating-Point Number Systems: floating point numbers have a sign, mantissa (M), base (B), and exponent (E). Floating point numbers are base 2 with a binary manitssa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits. In binary floating-point, the first bit of the mantissa is always 1 and therefore doesnt need to be stored. It is called the implicit leading one. The exponent needs to represent both positive and negative exponents. To do so, floating point uses a biased exponent, which is the original exponent plus a constant bias. 32-bit floating point uses a bias of 127. e.g representing $228_{10} = 1.11001_2 \times 2^7$



The IEEE floating point standard has special cases to represent zero,infinity and illegal results:

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Besides single-precision/single/float (32-bit floating-point) numbers there is also 64-bit double precision numbers (doubles) that provide greater precision and greater range.

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

The default rounding mode is round to nearest i.e if two numbers are equally near the one with a 0 in the least significant position of the fraction is chosen. A number overflows when its magnitude is too large to be represented (rounded to plus/minus infinity). Likewise a number underflows when it is too tiny to be represented(rounded to 0).
The addition of floating point numbers with the same sign is as follows:

1 Extract exponent and fraction bits

2 Prepend leading 1 to form the mantissa

3 Compare exponents

4 Shift smaller mantissa if necessary

5 Add mantissas

6 Normalize mantissa and adjust exponent if necessary

7 Round result.

8 Assemble exponent and fractzion back into floating point number

v

5.3 Sequential Building Blocks

Counters: An N-bit binary counter is a sequential arithmetic circuit with clock and reset inputs and an N-bit output Q. Reset initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

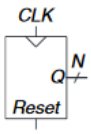


Figure 5.30 Counter symbol

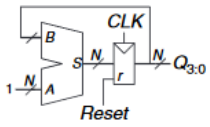


Figure 5.31 N-bit counter

Shift Registers: A shift register has a clock, a serial input S_{in} a serial output S_{out} and N parallel outputs $Q_{N-1:0}$. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}. Shift registers can be viewed as serial-to-parallel converters. (the input is provided serially i.e one at a time at S_{in} and after N cycles the past N inputs are available in parallel at Q. A shift register can be constructed from N flip-flops connected in series. A related circuit is a parallel-to serial converter that loads N bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both operations by adding a parallel input $D_{N-1:0}$ and a control signal Load. When Load is asserted, the flip-flops are loaded in parallel otherwise the shift register shifts normally.

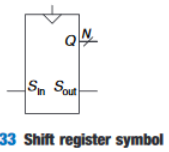
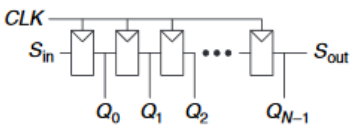
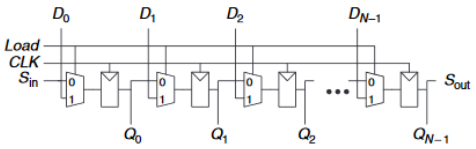


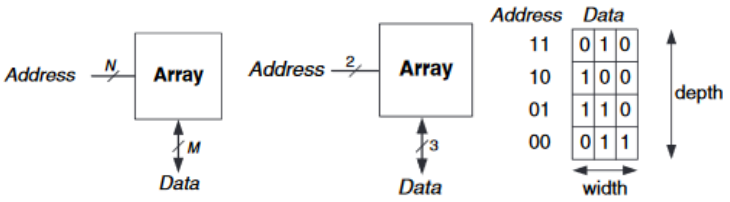
Figure 5.33 Shift register symbol





5.4 Memory Arrays

Overview: The memory is organized as a 2D array of memory cells. The memory reads or writes contents of one of the rows of the array which is specified by an **Address**. The value read/written is called **Data**. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is called a word. Hence the array contains 2^N M-bit words. The depth of an array is the number of rows, and the width is the number of columns, also called word size.



Memory arrays are build as an array of bit cells, each of which stores 1 bit of data. Bit cells are connected to a bitline and a wordline. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. We have the following operations:

- **Read:** The bitline is initially left floating. Then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1.
- **Write:** We drive the bitline to the desired value, then the wordline is turned ON, connecting the bitline to the stored bit. The bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

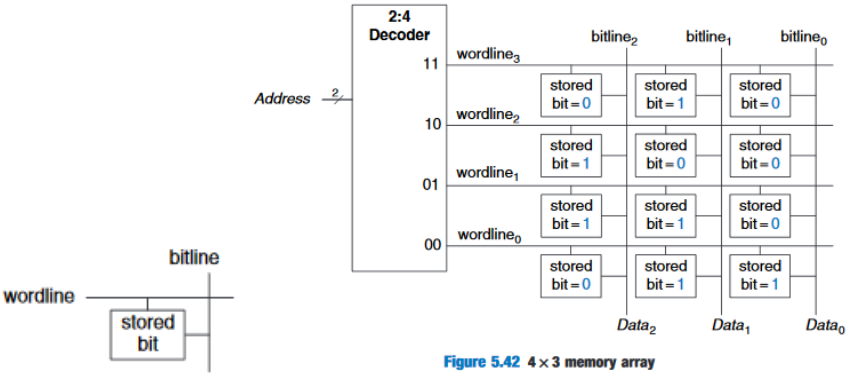


Figure 5.42 4 × 3 memory array

All memories have one or more ports. Each port gives read and/or write acces to one memory address. Multiported memories can access several addresses simultaneously.

Multiported memories can access several addresses simultaneously. Figure 5.43 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address *A1* onto the read data output *RD1*. Port 2 reads the data from address *A2* onto *RD2*. Port 3 writes the data from the write data input *WD3* into address *A3* on the rising edge of the clock if the write enable *WE3* is asserted.

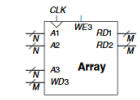
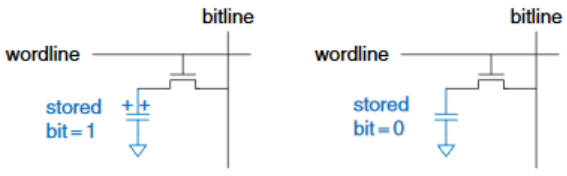


Figure 5.43 Three-ported memory

Memories are classified based on how they store bits in the bit cell.

- Random access memory (RAM): Volatile memory i.e it loses its data when the power is turned off. There are 2 major types of RAM:
  - dynamic RAM (DRAM): stores data as a charge on a capacitor
  - static RAM (SRAM): stores data using a pair of cross-coupled inverters
- Read only memory (ROM): Nonvolatile memory i.e it retains its data indefinitely, even without a power source. ROM takes a longer time to write to than RAM.

DRAM: stores a bit as the presence or absence of charge on a capacitor. The bit value is stored on the capacitor. The nMOS transistor behaves as a switch that connects/disconnects the capacitor from the bitline. When the wordline is asserted the nMOS turns ON and the stored bit value transfers to/from the bitline. The capacitor is dynamic because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND. Reading destroys the bit value stored on the capacitor, so the data word must be restored after each read. Even when DRAM is not read, the contents must be refreshed every few milliseconds, because the charge on the capacitor gradually leaks away.



SRAM: Static RAM is static because stored bits do not need to be refreshed. The data is stored on cross-coupled inverters. Each cell has 2 outputs, bitline and bitline. When the wordline is asserted, both nMOS transistors turn on, and data values are transferred to or from the bitlines. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

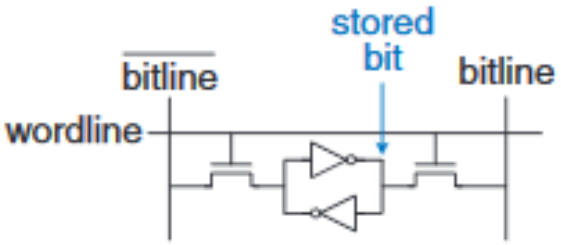


Figure 5.46 SRAM bit cell

Area and Delay: The following table shows a comparison of Flip-flops, SRAMs and DRAMs area and delay characteristics.

Table 5.4 Memory comparison

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

The data stored in a FF is available immediately at its output but takes atleast 20 transistors to build. (In General and increase in transistors means an increase in area, power and cost). DRAM has a longer latency than SRAM because its bitline is not actively driven by a transistor, it must wait for charge to move from the capacitor to the bitline. DRAM also has a lower throughput than SRAM, because it must refresh data periodically and after a read. The following DRAM's solve this problem:

- SDRAM (synchronous DRAM): Uses a clock to pipeline memory accesses
- DDR SDRAM (double data rate SDRAM): Uses both the rising and falling edges of the clock to access data, thus doubling the throughput for a given clock speed

Memory latency and throughput also depend on memory size; larger memories tend to be slower than smaller ones if all else is the same. (Best memory type for a particular design depends on the speed, cost and power constraints)

Register Files: A group of registers used to store temporary variables. The register file is usually built as a small multiported SRAM array, because its more compact than an array of FF's. E.g:

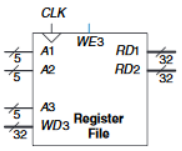
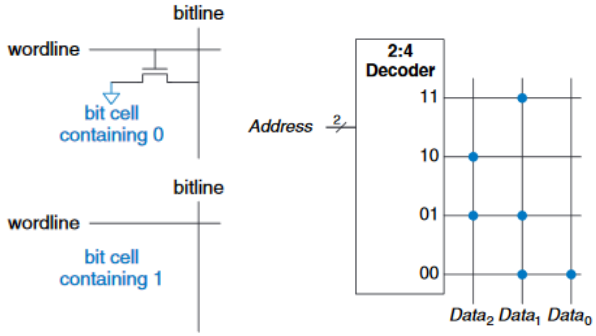
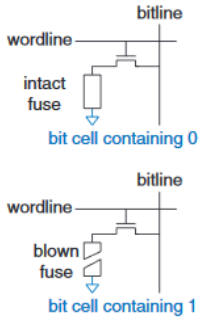


Figure 5.47 32 × 32 register file with two read ports and one write port

Read Only Memory (ROM): Stores a bit as the presence or absence of a transistor.To read the cell the bitline is weakly pulled HIGH. Then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent the bitline remains HIGH. Because the ROM bit cell is a combinational circuit it has no state to "forget" if power is turned off. The contents of a ROM can be indicated using dot notation. A dot at the intersection or a row (wordline) and a column (bitline) indicates that the data bit is 1.



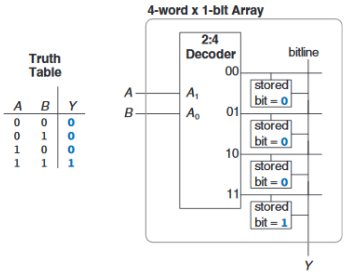
Conceptually ROMs can be built using two-level logic with a group of AND gates followed by a group of OR gates. (AND form the decoder, the ORs are connected to the dotted array entries) IN practice ROMs are built from transistors instead of logic gates to reduce size and cost. The contents of the ROM are specified during manufacturing by the presence or absence of a transistor in each bit cell. **Programmable ROM** places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground. E.g fuse-programmable ROM:



The user programs the ROM by applying a high voltage to selectively blow fuses. If the fuse is present the cell holds a 0, if its destroyed the transistor is disconnected from ground and the cell holds 1. (the fuse cannot be repaired once blown). Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND:

- **Erasable PROMs (EPROM):** replact the nMOS transistor and fuse with a floating-gate transistor which is not physically attached to any other wires. When suitable high voltages are applied electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline. UV light can be applied to the EPROM to knock off the electrons from the floating-gate turning the transistor off.
- **Electrically erasable PROMs (EEPROM):** and FLash memory include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM cells are more individually erasable.

Logic Using Memory Arrays: Memory arrays used to perform llogic are called **lookup tables**. Each address corresponds to an output value.

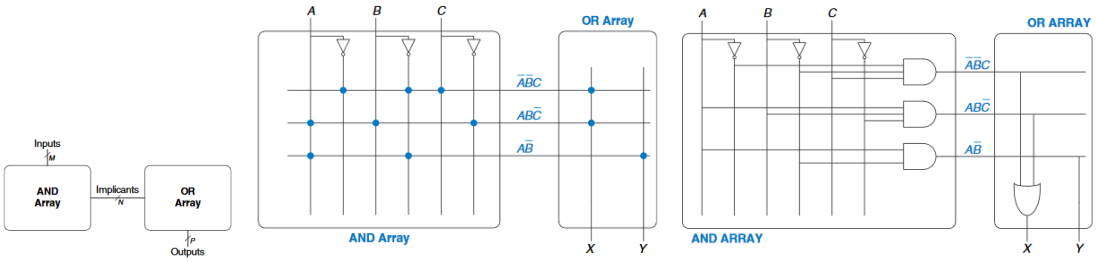


5.5 Logic Arrays

Like memory, gates can be organized into regular arrays known as logic arrays. If the connections are made programmable they can be configured to perform any function without the user having to connect wires in specific ways. There are two types of logic arrays:

Programmable Logic Array (PLAs): implement two level combinational logic in sum-of-products form. PLAs are built from an AND array followed by an OR array. The inputs drive an AND array, which produces implicants, which are then ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants and P outputs. ROMs can be viewed as a special case of PLAs. A 2^M-word $\times N$-bit ROM is simply an $M \times 2^M \times N$-bit PLA. The decoder behaves

as an AND plane that produces all 2^M minterms. The ROM array behaves as an OR plane that produces the outputs. If the function does not depend on all 2^M minterms, a PLA is likely to be smaller than a ROM.



Field Programmable Gate Array (FPGA): An array of reconfigurable gates. With software programming tools, a user can implement designs on the FPGA using either an HDL or a schematic. They are more powerful and flexible than PLAs because:

- They can implement both combinational and sequential logic
- They can implement multi-level logic functions
- integrate useful features like built-in multipliers, high-speed I/O, data converters, large RAM arrays and processors

FPGA's are built as an array of configurable logic elements (LE's) which can be configured to perform combinational or sequential functions

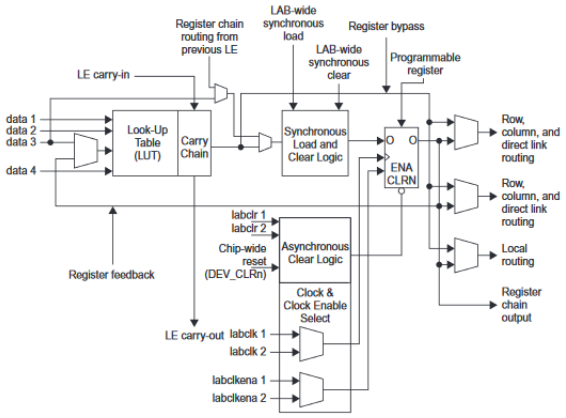


Figure 5.58 Cyclone IV Logic Element (LE)

The FPGA is configured by specifying the contents of the lookup tables and the select signals for the multiplexers.

Chapter 6

Architecture

The architecture is the programmers view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). The words in a computers language are called instructions and the computers vocabulary is called the instruction set. Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language. Microprocessors are digital systems that read and execute machine language instructions. (Programmers represent the instructions in a symbolic format called assembly language). A computer architecture does not define the underlying hardware implementation i.e multiple can exist for a single architecture. The four principles of good architecture:

- Simplicity favors regularity
- Make the common case fast
- Smaller is faster
- Good design demands good compromises

6.1 Assembly Language

The humand-readable representation of the computers native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate.

Instructions: The first part of the assembly instruction is called the **mnemonic** and indicates what operation to perform. The operation is performed on the **source operands** and the result is written to the **destination operand**. Instructions with a consistent number of operands are easier to encode and handle in hardware, more complex high-level code translates into multiple MIPS instructions.

High-Level Code	MIPS Assembly Code
`a = b + c;`	`add a, b, c`

High-Level Code	MIPS Assembly Code
`a = b - c;`	`sub a, b, c`

High-Level Code	MIPS Assembly Code

`a = b + c - d;  // single-line comment`
`/* multiple-line`
`comment */`
	`sub t, c, d  # t = c - d`
	`add a, b, t  # a = b + t`

In assembly language, only single-line comments are used. They begin with # and continue until the end of teh line. The MIPS instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small and fast. Hence MIPS is a **reduced instruction set computer (RISC)** architecture. Architectures with many complex instructions are **complex instruction set computers(CISC)**. In a CISC machine more bits are needed to encode operations, hence even if complex instructions are used rarely, they add overhead to all instructions, even the simple ones.

Operands: An instruction operates on operands. Operands can be stored in registers or memory or may be constants stored in the instruction itself. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory which is large but slow.

Registers: Most architectures specify a small number of registers that hold commonly used operands in order to access operands quickly. MIPS uses 32 registers, called the register set or register file. The fewer the registers, the faster they can be accessed. A small register file is typically built from a small SRAM array. The SRAM array uses a small decoder and bitlines conncected to relatively few memory cells, so it has a shorter critical path than a large memory does.
MIPS register names are preceded by the $ sign. MIPS generally stores variables in 18 of the 32 registers: $\$s0 - \$s7$, and $\$t0 - \$t9$. Register names beginning with $\$s$ are called saved registers, these registers store variables such as a,b,c. Register names beginning with $\$t$ are called temporary registers and are used for storing temporary variables.

High-Level Code	MIPS Assembly Code
`a = b + c;`	`# $s0 = a, $s1 = b, $s2 = c` `add $s0, $s1, $s2  # a = b + c`

High-Level Code	MIPS Assembly Code

`a = b + c - d;`	`# $s0 = a, $s1 = b, $s2 = c, $s3 = d`
	`sub $t0, $s2, $s3  # t = c - d`
	`add $s0, $s1, $t0  # a = b + t`

The Register Set: MIPS architecture defines 32 registers, each having a name and a number ranging from 0 to 31:

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Memory: When compared to the register file, memory has many data locations, but accessing it takes a longer amount of time. For this reason commonly used variables are kept in registers. By using a combination of memory and registers, a program can access a large amount of data quickly. The MIPS architecture uses 32-bit memory addresses and 32-bit data words. MIPS uses **byte-addressable memory** i.e each byte in memory has a unique address. By convention memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top. The load word instruction lw, is used to read a data word from memory into a register. The lw insturction specifies the effective address in memory as the sum of a base address and an offset. The base address (written in parantheses in the instruction) is a register. The offset is a constant(written before the parantheses). The store word instruction, sw is used to write a data word from a register into memory

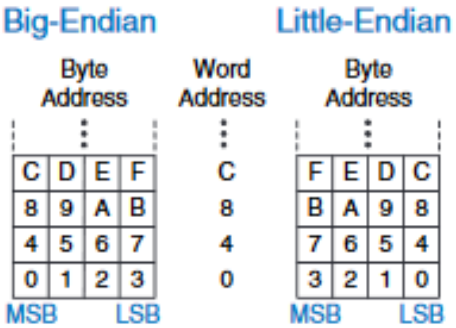
Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw  $s7, 5($0)    # write $s7 to memory word 5
```

The above examples use word addressable memory (MIPS is byte addressable). Each data byte ahas a unique address. A 32-bit word consists of four 8-bit bytes. Byte-addressable memories are organized in a big-endian or little-endian fashion. In both formats, the MSB is on the left and the LSB is on the right. In big-endian machines, bytes are numbered starting with 0 at the most significant (Big) end. In little-endian machines, bytes are numbered starting with 0 at the least significant (little) end. Word addresses are the same in both formats and refer to the same four bytes.



MIPS Assembly Code

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

In the MIPS architecture, word addresses for lw and sw must be word aligned (address divisible by 4)

Constants/immediates: Load word and store word use constants called immediates. The get their name because the values are immediately available from the instruction and do not require a reigster or memory access. Add immediate **addi** adds the immediate specified in the instruction to a value in a register. The immediate specified in an instruction is a 16-bit two's complement number in the range [-32,768, 32,767]

High-Level Code

```
a = a + 4;
b = a - 12;
```

MIPS Assembly Code

```
# $s0 = a, $s1 = b
  addi $s0, $s0, 4      # a = a + 4
  addi $s1, $s0, -12    # b = a - 12
```

6.2 Machine Language

A program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called machine language which digital circuits understand. MIPS defines three instruction formats:

- R-type : operates on three registers

- I-type : operates on two registers and a 16-bit immediate
- J-type : operate on one 26-bit immediate

R-Type Instructions: R-type is short for register-type. These instructions use three registers as operands: two as sources and one as a destination. The 32-bit instruction has 6 fields:

- op:
- rs: operand (source register)
- rt: operand (source register)
- rd: operand (destination register)
- shamt: only used in shift operations the binary value stored here indicates the amount to shift
- funct:

The operation the instruction performs is encoded in op(opcode) and funct(function). All R-type instructions have an opcode of 0, hence the specific R-type operation is determined by funct.

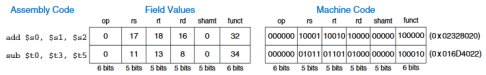


Figure 6.6 Machine code for R-type instructions

I-type Instructions: I-type is short for immediate-type. These instructions use two register operands and one immediate operand. The 32-bit instruction has four fields: op,rs,rt and imm. The first three are like those of R-type. The imm field holds the 16-bit immediate. The operation is determined by the opcode. rs and imm are always used as source operands and rt used as a source or destination dependant on the instruction.

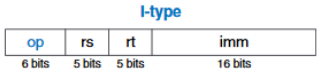


Figure 6.8 I-type instruction format

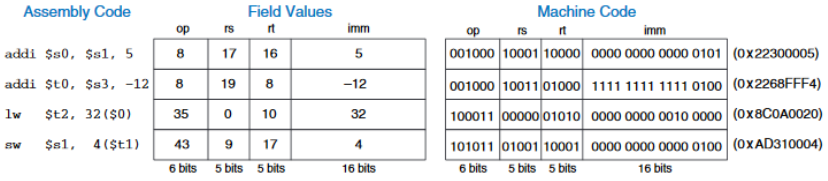
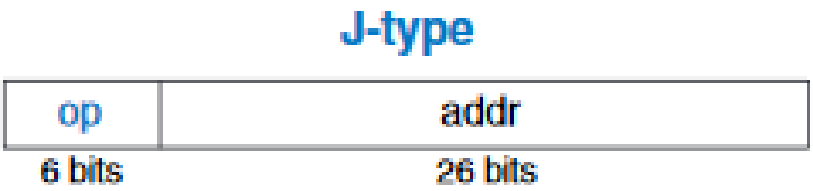


Figure 6.9 Machine code for I-type instructions

J-Type Instructions: J-type is short for jump-type. This format is only used with jump instructions. Like other formats J-type instructions begin with a 6-bit opcode. The remaining bits are used to specify an address addr.



Interpreting Machine Language Code: To interpret machine language the fields of each 32-bit instruction word must be deciphered. Different instructions use different formats, but all start with a 6-bit opcode field. If the op code is 0, the instruction is R-type; otherwise it is I-type or J-type. The opcode determines how to interpret the rest of the bits.

Stored Program: A program written in machine language is a series of 32-bit numbers representing instructions which can be stored in memory. This allows computers to run different programs fast and efficiently without reconfiguring/rewiring hardware, we only need to write the new program to memory. Instead of dedicated hardware the stored program offers general purpose computing. Instructions in a stored program are retrieved (fetched) from memory and executed by the processor. To run (execute) the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the **program counter (PC)**. To execute the code the OS sets the PC to the starting address, the processor reads the instruction at that memory address and executes the instruction. The processor then increments the PC by 4 (MIPS is byte-addressable) and executes that instructions, and repeats.

Architectural State: Holds the state of a program. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, then restore the state such that the program continues properly, unaware that it was ever interrupted.

6.3 Programming

Arithmetic/Logical Instructions:

- **Logical Instructions:** MIPS logical operations include "and,or,xor, and nor". These R-type instr. operate bit by bit on two source registers and write the result to the destination register.

	Source Registers							
$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
and $s3, $s1, $s2	$s3	0100	0110	1010	0001	0000	0000	0000	0000
or $s4, $s1, $s2	$s4	1111	1111	1111	1111	1111	0000	1011	0111
xor $s5, $s1, $s2	$s5	1011	1001	0101	1110	1111	0000	1011	0111
nor $s6, $s1, $s2	$s6	0000	0000	0000	0000	0000	1111	0100	1000

The "and" instruction is useful for masking bits (forcing unwanted bits to 0). The "or" instruction is useful for combining bits from two registers. MIPS does not provide a NOT instr. but A NOR $0 = NOT A, hence NOR can substitute for NOT. Logical operations can also operate on immediates. These I-type instructions are "andi,ori, and xori". "nori" is not provided as the same functionality can be easily implemented using the other instructions.

- **Shift Instruction:** Shift the value in a register left or right by up to 31 bits. MIPS shift operations are "sll" (shift left logical), "srl" (shift right logical), and "sra" (shift right arithmetic). MIPS also has variable-shift instructions: "sllv, srlv and srav". Variable shift assembly instrcutions are of the form "sllv rd. rt. rs." The order of rt and rs is reversed from most R-type instructions. rt holds the value to be shifted, and the 5 least significant bits of rs give the amount to shift, the result is placed in rd. The shamt field is ignored and should be all 0's

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
sllv $s3, $s1, $s2	0	18	17	19	0	4	000000	10010	10001	10011	00000	000100	(0x02519804)
srlv $s4, $s1, $s2	0	18	17	20	0	6	000000	10010	10001	10100	00000	000110	(0x0251A006)
srav $s5, $s1, $s2	0	18	17	21	0	7	000000	10010	10001	10101	00000	000111	(0x0251A807)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.18 Variable-shift instruction machine code

	Source Values							
$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Assembly Code	Result								
sllv $s3, $s1, $s2	$s3	0000	0100	0000	0010	1010	1000	0000	0000
srlv $s4, $s1, $s2	$s4	0000	0000	1111	0011	0000	0100	0000	0010
srav $s5, $s1, $s2	$s5	1111	1111	1111	0011	0000	0100	0000	0010

Figure 6.19 Variable-shift operations

- **Generating Constants:** The addi instruction is helpful for assigning 16-bit constants. To assign 32-bit constants, use a load upper immediate instruction "lui" followed by an or immediate "ori" instruction. (lui loads 16-bit immediate into the upper half of a register and sets the lower half to 0, ori merges a 16-bit immediate into the lower half)

High-Level Code	MIPS Assembly Code
`int a = 0x4f3c;`	`# $s0 = a` `addi $s0, $0, 0x4f3c  # a = 0x4f3c`

High-Level Code	MIPS Assembly Code

`int a = 0x6d5e4f3c;`	`# $s0 = a`
	`lui $s0, 0x6d5e  # a = 0x6d5e0000`
	`ori $s0, $s0, 0x4f3c  # a = 0x6d5e4f3c`

- **Multiplication and Division Instructions:** The MIPS architecture has two special purpose registers "hi" and "lo", which are used to hold the results of multiplication and division. For multiplication the 32 most significant bits of the product are placed in hi and the 32 least significant bits are placed in lo. For division the quotient is placed in lo and the remainder is placed in hi.

Branching: To sequentially execute instructions, the PC increments by 4 after each instruction. Branch instructions modify the PC to skip over sections of code or to repeat previous code. There are two types of branch instructions:

- **Conditional Branches:** Conditional branch instructions perform a test and branch only if the test is TRUE. MIPS instruction set has two conditional branch instructions:
  - **branch if equal (beq):** branches when the values in two registers are equal
  - **branch if not equal (bne):** branches when the values in two registers are not equal

branches are written as "beq rs, rt, imm" where rs is the first source register.

MIPS Assembly Code

```
addi  $s0, $0, 4        # $s0 = 0 + 4 = 4
addi  $s1, $0, 1        # $s1 = 0 + 1 = 1
sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
beq   $s0, $s1, target  # $s0 == $s1, so branch is taken
addi  $s1, $s1, 1       # not executed
sub   $s1, $s1, $s0     # not executed

target:
add   $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

MIPS Assembly Code

```
addi  $s0, $0, 4        # $s0 = 0 + 4 = 4
addi  $s1, $0, 1        # $s1 = 0 + 1 = 1
sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
bne   $s0, $s1, target  # $s0 == $s1, so branch is not taken
addi  $s1, $s1, 1       # $s1 = 4 + 1 = 5
sub   $s1, $s1, $s0     # $s1 = 5 - 4 = 1

target:
add   $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

- **Unconditional Branching:** Also called jumps, always branch. There are 3 types of jump instructions:
  - **jump (j):** Jumps directly to the instruction at the specified label.
  - **jump and link (jal):** similar to j but is used by functions to save a return address.
  - **jump register (jr):** Jumps to an address held in a register

MIPS Assembly Code

```
addi  $s0, $0, 4      # $s0 = 4
addi  $s1, $0, 1      # $s1 = 1
j     target          # jump to target
addi  $s1, $s1, 1     # not executed
sub   $s1, $s1, $s0   # not executed

target:
add   $s1, $s1, $s0   # $s1 = 1 + 4 = 5
```

MIPS Assembly Code

```
0x00002000  addi  $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004  jr    $s0              # jump to 0x00002010
0x00002008  addi  $s1, $0, 1       # not executed
0x0000200c  sra   $s1, $s1, 2      # not executed
0x00002010  lw    $s3, 44($s1)     # executed after jr instruction
```

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses. MIPS assembly labels are followed by a colon and cannot use reserved words.

Conditional Statements:

- **If statements:** An if statement executes a block of code, the if block, only when a condition is met. The assembly code for the if statement tests the opposite condition of the one in the high-level code.

High-Level Code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, L1    # if i != j, skip if block
  add $s0, $s1, $s2   # if block: f = g + h
L1:
  sub $s0, $s0, $s3   # f = f - i
```

- **If/Else Statements:** if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met the if block is executed. Otherwise the else block is executed.

High-Level Code

```
if (i == j)
  f = g + h;

else
  f = f - i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else  # if i != j, branch to else
  add $s0, $s1, $s2   # if block: f = g + h
  j   L2              # skip past the else block
else:
  sub $s0, $s0, $s3   # else block: f = f - i
L2:
```

- **Switch/Case Statements:** execute one of several blocks of code depending on the conditions. If no conditions are met, the default block is executed. A case statement is equivalent to a series of nested if/else statements.

High-Level Code

```
switch (amount) {
  case 20:   fee = 2; break;

  case 50:   fee = 3; break;

  case 100:  fee = 5; break;

  default: fee = 0;
}

// equivalent function using if/else statements
  if      (amount == 20)  fee = 2;
  else if (amount == 50)  fee = 3;
  else if (amount == 100) fee = 5;
  else                    fee = 0;
```

MIPS Assembly Code

```
# $s0 = amount, $s1 = fee
case20:
  addi $t0, $0, 20        # $t0 = 20
  bne  $s0, $t0, case50   # amount == 20? if not,
                          # skip to case50
  addi $s1, $0, 2         # if so, fee = 2
  j    done               # and break out of case
case50:
  addi $t0, $0, 50        # $t0 = 50
  bne  $s0, $t0, case100  # amount == 50? if not,
                          # skip to case100
  addi $s1, $0, 3         # if so, fee = 3
  j    done               # and break out of case
case100:
  addi $t0, $0, 100       # $t0 = 100
  bne  $s0, $t0, default  # amount == 100? if not,
                          # skip to default
  addi $s1, $0, 5         # if so, fee = 5
  j    done               # and break out of case
default:
  add  $s1, $0, $0        # fee = 0
done:
```

Loops:

- **while loops:** repeatedly execute a block of code until a condition is not met. Like if/else statements, the assembly code for while loops tests the opposite condition of the one given in the high-level code.

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128)
{
  pow = pow * 2;
  x = x + 1;
}
```

MIPS Assembly Code

```
# $s0 = pow, $s1 = x
  addi $s0, $0, 1      # pow = 1
  addi $s1, $0, 0      # x = 0

  addi $t0, $0, 128    # t0 = 128 for comparison
while:
  beq  $s0, $t0, done  # if pow == 128, exit while loop
  sll  $s0, $s0, 1     # pow = pow * 2
  addi $s1, $s1, 1     # x = x + 1
  j    while
done:
```

- **For loops:** repeatedly execute a block of code until a condition is not met. For loops add support for a loop variable, which typically keeps track of the number of loop executions. The initialization code executes before the for loop begins. The condition is tested at the beginning of each loop. If the condition is not met the loop exits. The loop operation executes at the end of each loop.

High-Level Code

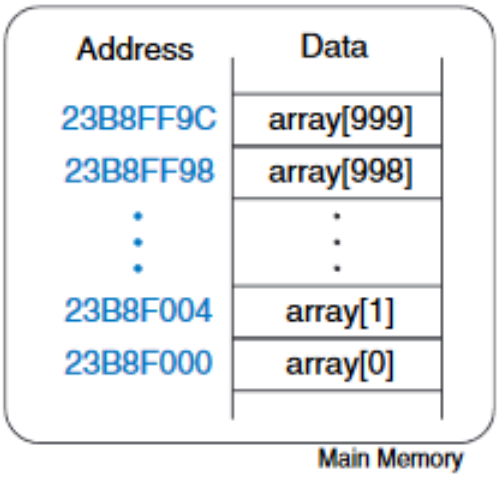
```
int sum = 0;

for (i = 0; i != 10; i = i + 1) {
  sum = sum + i ;

}

// equivalent to the following while loop
int sum = 0;
int i = 0;
while (i != 10) {
  sum = sum + i;
  i = i + 1;
}
```

MIPS Assembly Code

```
# $s0 = i, $s1 = sum
  add  $s1, $0, $0     # sum = 0
  addi $s0, $0, 0      # i = 0
  addi $t0, $0, 10     # $t0 = 10
for:
  beq  $s0, $t0, done  # if i == 10, branch to done
  add  $s1, $s1, $s0   # sum = sum + i
  addi $s0, $s0, 1     # increment i
  j    for
done:
```

- **Magnitude Comparison:** MIPS provides a set less than instruction "slt", for magnitude comparison. slt sets rd to 1 when rs ¡ rt. Otherwise rd is 0.

Arrays: Useful for accessing large amounts of similar data. An array is organized as sequential data addresses in memory. Each array element is identified by a number called its index. The number of elements in the array is called the size of the array.



High-Level Code

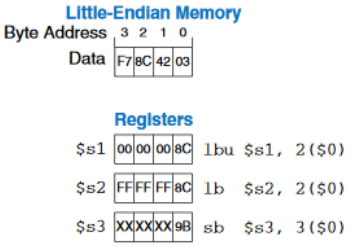
```
int i;
int array[1000];

for (i = 0; i < 1000; i = i + 1)

  array[i] = array[i] * 8;
```

MIPS Assembly Code

```
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x2388        # $s0 = 0x23880000
  ori  $s0, $s0, 0xF000   # $s0 = 0x2388F000
  addi $s1, $0, 0         # i = 0
  addi $t2, $0, 1000      # $t2 = 1000
loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if not, then done
  sll  $t0, $s1, 2        # $t0 = i*4 (byte offset)
  add  $t0, $t0, $s0      # address of array[i]
  lw   $t1, 0($t0)        # $t1 = array[i]
  sll  $t1, $t1, 3        # $t1 = array[i] * 8
  sw   $t1, 0($t0)        # array[i] = array[i] * 8
  addi $s1, $s1, 1        # i = i + 1
  j    loop               # repeat
done:
```

Numbers in the range [-128,127] can be stored in a single byte rather than an entire word. English characters are often represented by bytes. The language C uses the type char to represent a byte or a character. MIPS provides load byte and store byte instructions to manipulate bytes or characters of data:

- **load byte unsigned:** zero extends the byte to fill 32-bit register
- **load byte:** sign-extends the byte to fill the entire 32-bit register
- **store byte:** stores the least significant byte of the 32-bit register into the specified byte address in memory.



#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

A series of characters is called a string. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C the null character (0x00) singifies the end of a string.

Function Calls: Functions (procedures) allow frequently accessed code to be reused hence make a program more modular and readable. Functions have inputs (arguments) and an output (return value). When one function (caller) calls another (callee), both functions must agree on where to put the arguments and the return value. In MIPS the caller convetntionally places up to four arguments in register $\$a0 - \$a3$ before making the function call, and the callee places the return value in registers $\$v0 - \$v1$ before finishing, hence both functions know where to find the arguments and return value.
The callee must not interfere with the function of the caller (must not trample on any registers or memory needed by the caller) The caller stores the return address in $\$ra$ at the same time it jumps to the callee using the "jump and link" instruction (jal). The callee must not overwrite any architectural state or memory that the caller is depending on. (it must leave the saved registers $\$s0 - \$s7, \$ra$ and the stack (portion of memory used for temporary variables) unmodified.

Function Calls and Returns: MIPS uses "jal" to call a function and the "jump register (jr)" to return from a function. jal performs two operations:

1. stores the address of the next instruction (i.e instruction after jal) in the return address register ($\$ra$)
2. Jumps to taget instruction

High-Level Code

```
int main() {
  simple();
  . . .
}
// void means the function returns no value
void simple() {
  return;
}
```

MIPS Assembly Code

```
0x00400200 main:    jal simple  # call function
0x00400204          . . .


0x00401020 simple:  jr $ra      # return
```

Input Arguments and Return Values: A funct5ion that returns a 64-bit value such as double-precision, floating point number, uses both return register $\$v0, \$v1$. When a function with more than four arguments is called, the additional input arguments are placed on the stack.

High-Level Code

```
int main()
{
  int y;

  . . .

  y = diffofsums(2, 3, 4, 5);

  . . .
}

int diffofsums(int f, int g, int h, int i)
{
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

MIPS Assembly Code

```
# $s0 = y
main:
  . . .
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call function
  add  $s0, $v0, $0  # y = returned value
  . . .

# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

The Stack: The stack is memory that is used to save local variables within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. The stack is a last-in-first-out (LIFO) queue. The last item pushed onto the stack is the first one that can be popped off. Each function may allocate stack space to store local variables but must deallocate it before returning. The MIPS stack grows down in memory i.e it expands to lower memory addresses when a program needs more scratch space. $\$sp$ is a special MIPS register that points to the top of the stack. (A pointer is a fancy name for a memory address). One important uses of the stack is to save and restore registers that are used by a function. A function saves registers on the stack before it modifies them, then restores them from the stack before it returns. It performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

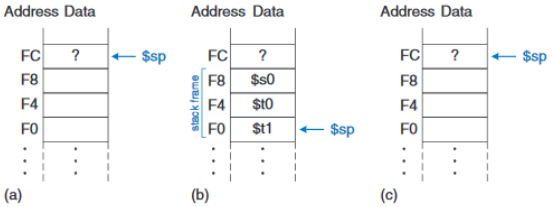


Figure 6.25 The stack (a) before, (b) during, and (c) after `diffofsums` function call

MIPS Assembly Code

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12 #  make space on stack to store three registers
  sw   $s0, 8($sp)   #  save $s0 on stack
  sw   $t0, 4($sp)   #  save $t0 on stack
  sw   $t1, 0($sp)   #  save $t1 on stack
  add  $t0, $a0, $a1 #  $t0 = f + g
  add  $t1, $a2, $a3 #  $t1 = h + i
  sub  $s0, $t0, $t1 #  result = (f + g) - (h + i)
  add  $v0, $s0, $0  #  put return value in $v0
  lw   $t1, 0($sp)   #  restore $t1 from stack
  lw   $t0, 4($sp)   #  restore $t0 from stack
  lw   $s0, 8($sp)   #  restore $s0 from stack
  addi $sp, $sp, 12  #  deallocate stack space
  jr   $ra           #  return to caller
```

Preserved Registers: MIPS divides registers into preserved and nonpreserved categories:

- **preserved registers: (callee save)** include $\$s0 - \$s7$ hence the name saved
- **nonpreserved register:(caller save)** include $\$t0 - \$t9$ hence the name temporary

A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely. If the caller function is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward.

Preserved	Nonpreserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Return address: `$ra`	Argument registers: `$a0–$a3`
Stack pointer: `$sp`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Recursive Function Calls: A function that does not call others is called a **leaf function**, otherwise its called a **nonleaf** function. A recursive function is a nonleaf function that calls itself. E.g factorial function: factorial(n) = n * factorial(n-1)

High-Level Code

```
int factorial(int n) {


 if (n <= 1)
   return 1;



 else
   return (n * factorial(n - 1));
}
```

MIPS Assembly Code

```
0x90   factorial:   addi $sp, $sp, -8    # make room on stack
0x94                sw   $a0, 4($sp)     # store $a0
0x98                sw   $ra, 0($sp)     # store $ra
0x9C                addi $t0, $0, 2      # $t0=2
0xA0                slt  $t0, $a0, $t0   # n <= 1 ?
0xA4                beq  $t0, $0, else   # no: goto else
0xA8                addi $v0, $0, 1      # yes: return 1
0xAC                addi $sp, $sp, 8     # restore $sp
0xB0                jr   $ra             # return
0xB4        else:   addi $a0, $a0, -1    # n = n - 1
0xB8                jal  factorial       # recursive call
0xBC                lw   $ra, 0($sp)     # restore $ra
0xC0                lw   $a0, 4($sp)     # restore $a0
0xC4                addi $sp, $sp, 8     # restore $sp
0xC8                mul  $v0, $a0, $v0   # n * factorial(n-1)
0xCC                jr   $ra             # return
```

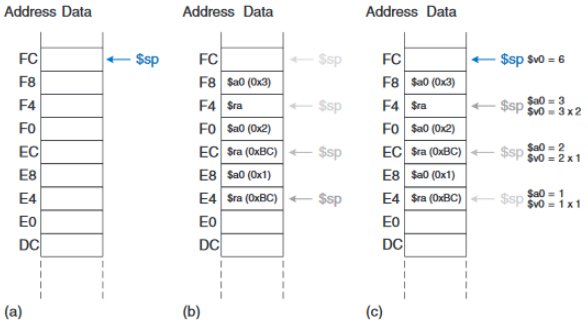


Figure 6.26 **Stack during `factorial` function call when `n` = 3: (a) before call, (b) after last recursive call, (c) after return**

Additional Arguments and Local Variables: By MIPS convention if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above $*sp*. The caller must expand its stack to make room for the additional arguments. A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in the saved registers, if there are too many they can also be stored in the functions stacik frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.4 Addressing Modes:

MIPS uses five addressing modes, the first three define modes of reading, the last two define modes of writing the program counter.

- **Register-Only Addressing:** Register-only addressing uses registers for all souce and destination operands. All R-type instructions use register-only addressing.
- **Immediate Addressing:** Uses the 16-bit immediate along with registers as operands. Some I-type instructions e.g addi, lui use immediate addressing.
- **Base Addressing:** The effective address of the memory operand is found by adding the base address in register rs to the sign extended 16-nit offset found in the immediate field. Memory access instructions use base addressing e.g load word and store word
- **PC-Relative Addressing:** Conditional branch instructions use PC-relative addressing to specify the new value of the PC if the branch is taken. The signed offset in the immediate field is added to the PC to obtain the new PC. The **branch target address (BTA)** is the address of the next instruction to execute if the branch is taken. The processor calculates teh BTA from the instruction by sign-extending the 16-bit immediate, multiplying it by 4 and adding it to PC+4

Example 6.8 **CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING**

Calculate the immediate field and show the machine code for the branch not equal (`bne`) instruction in the following program.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Solution:

Figure 6.29 shows the machine code for the `bne` instruction. Its branch target address, 0x40, is 6 instructions behind `PC` + 4 (0x58), so the immediate field is –6.

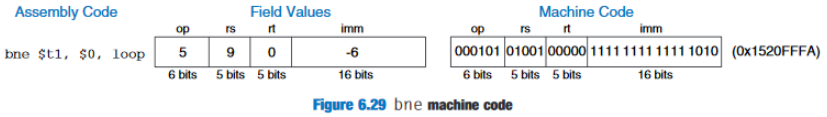
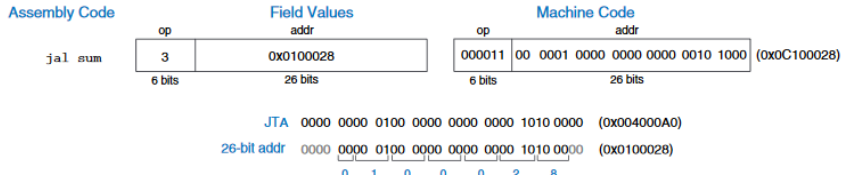


Figure 6.29 `bne` **machine code**

- **Pseudo-Direct Addressing:** In direct addressing, an address is specified in the instruction. J-type instruction encoding does not have enough bits to specify a full 32-bit (jump-target address). 6 bits are used for the opcode hence 26-bits are left to encode JTA. The two least significant bits $JTA_{1:0}$ should always be 0, because instructions are word aligned. the next 26 bits $JTA_{27:2}$ are taken from the addr field of the instruction. The four most significant bits $JTA_{31:28}$ are obtained from the four most significant bits of PC+4. Because the four most significant bits of the JTA are taken from PC+4, the jump range is limited.



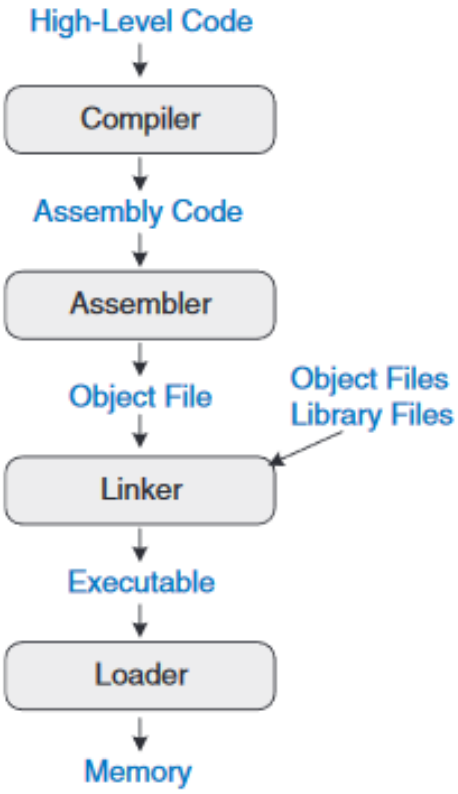
6.5 Compiling,Assembling and Loading:

Memory map: Defines where code data, and stack memory are located. With 32-bit addr4esses the MIPS address space spans $2^{32} = 4$ Gigabytes. Word addresses are divisible by 4 and range from 0 to 0xFFFFFFC. The MIPS architecture divides the address space into four parts (segments):

- **Text Segment:** stores the machine language program. It has 256MB of space. The four most significant bits of the address in the text space are all 0, so the j instruction can directly jump to any address in the program.
- **Global Data segment:** Stores global variables (i.e variables which can be seen and accessed by all functions in a program). Global variables are defined at start-up before program execution. The global data segment has 64KB of space for global variables. Global variables are accessed using the global pointer $\$gp$ (initialized to 0x100080000) $\$gp$ does not change during program execution. Any global variable can be accessed with a 16-bit positive/negative offset from $\$gp$. The offset is known at assembly time so the variables can be efficiently accessed using base addressing mode with constant offsets.
- **Dynamic Data Segment:** holds the stack and the heap. The data is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program. It has a space of almost 2GB. The heap stores data that is allocated by the program during runtime. (In Java "new" is used to allocate memory). Heap data can be used and discarded in any order, it grows upward from the bottom of the dynamic data segment. If the stack and heap ever grow into each other the programs data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.
- **Reserved Segment:** Used by the OS and cannot directly be used by the program. Part of the reserved memory is used for interrupts and for memory mapped I/O

Translating and starting a program: Steps required to translate a program from high-level language into machine language and to start execution:

1. High-Level code is compiled into assembly code
2. Assembly code is assembled into machine code in an object file
3. Linker combines machine code with object code from libraries and other files to produce an entire executable program
4. Loader loads the program into memory and starts execution



Step 1: Compilation: A compiler translates high-level code into assembly language. E.g

High-Level Code

```
int f, g, y; // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi $sp, $sp, -4  # make stack frame
  sw   $ra, 0($sp)   # store $ra on stack
  addi $a0, $0, 2    # $a0 = 2
  sw   $a0, f        # f = 2
  addi $a1, $0, 3    # $a1 = 3
  sw   $a1, g        # g = 3
  jal  sum           # call sum function
  sw   $v0, y        # y = sum(f, g)
  lw   $ra, 0($sp)   # restore $ra from stack
  addi $sp, $sp, 4   # restore stack pointer
  jr   $ra           # return to operating system

sum:
  add  $v0, $a0, $a1 # $v0 = a + b
  jr   $ra           # return to caller
```

The .data and .text keywords are assembler directives that indicate where the text and data segments begin. Labels are used for global variables (f,g,y) The storage location will be determined by the assembler.

Step 2: Assembling: The assembler turns the assembly language code into an object file containing machine language code. The assembler makes two passes though the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names. The names and addresses of the symbols are kept in a symbol table (filled in after the first pass, when the addresses of labels are known). On the second pass, the assembler produces the machine language code. The machine language code and symbol table are stored in the object file. (left picture is code after first pass)

```
0x00400000 main: addi $sp, $sp, -4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

Table 6.4 Symbol table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

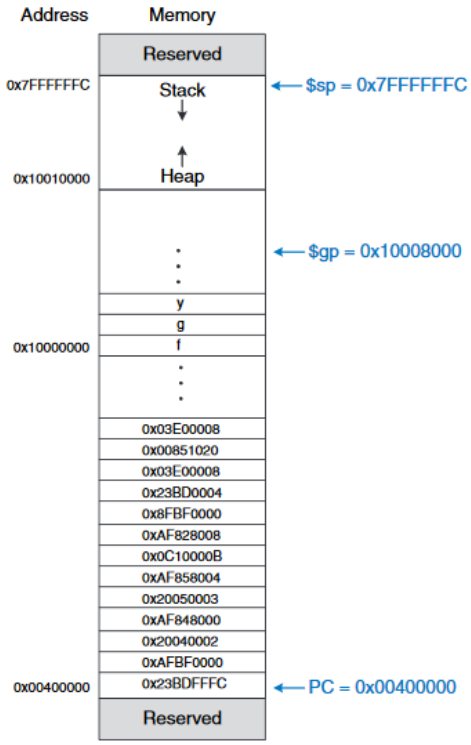
Linking: If a file of high-level code is not changed, the associated object file need not be updated. The job of the linker is to combine all of the object files into one machine language file called the **executable**. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses information in the symbol tables to adjust the addresses of global variables and of labels that are relocated. The below image is an example of an executable file. It is split into three parts:

- Executable file header: Reports the text size (code size) and data size (amount of globally declared data) in bytes.
- Text segment: Gives the instructions in the order that they are stored in memory
- Data segment: Gives the address of each global variable.

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0($sp)
addi $a0, $0, 2
sw   $a0, 0x8000($gp)
addi $a1, $0, 3
sw   $a1, 0x8004($gp)
jal  0x0040002C
sw   $v0, 0x8008($gp)
lw   $ra, 0($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Loading: The OS loads a program by reading the text segment of the executable file from a storage device (usually hard disk) into the text segment of memory. The OS sets $*gp* to 0x10008000 (the middle of the global data segment) and $*sp* to 0x7FFFFFFC (the top of the dynamic data segment), then performs a jal 0x00400000 to jump to the beginning of the program


Address
Memory
Reserved
0x7FFFFFFC
Stack
$sp = 0x7FFFFFFC
0x10010000
Heap
$gp = 0x10008000
y
g
0x10000000
f
0x03E00008
0x00851020
0x03E00008
0x23BD0004
0x8FBF0000
0xAF828008
0x0C10000B
0xAF858004
0x20050003
0xAF848000
0x20040002
0xAFBF0000
0x00400000
0x23BDFFFC
PC = 0x00400000
Reserved

Chapter 7

Microarchitecture

Microarchitecture is the connection between logic and architecture. It is the specific arrangement of registers,ALUs, finite state machines, memories and other logic building blocks needed to implement an architecture. A particular architecture may have many different microarchitectures, they all run the same programs but their internal designs vary widely

7.1 Intro

Architectural State and Instruction Set: A computer architecture is defined by its instructions set and architectural state. For the MIPS processor this consists of the program counter and the 32 registers. Any MIPS microarchitecture must contain all of this state. Based on the current architectural state the processor executes a particular instruction with a particular set of data to produce a new architectural state. The following examples will handle a subset of the MIPS instruction set:

- R-type: add,sub,and,or,slt
- Memory instructions: lw, sw
- Branches: beq

Design Process: We divide the microarchitectures into two interacting parts:

- **Data Path:** operates on words of data. It contains structures such as memories,registers, ALU's and multiplexers. Since MIPS is a 32.-bit architecture we will use 32-bit datapath.
- **Control:** The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction i.e the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system:

- Start with hardware containing the state elements (memories and architectural state i.e PC and registers)
- Add blocks of combinationalo logice between state elements to compute the new state based on the curent state.
- Partition the overall memory into two smaller memories, one containing instructions and the other containing data (Instructions are read from part of memory and lw, sw read/write data from another part of memory)

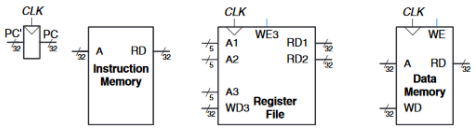


Figure 7.1 State elements of MIPS processor

Narrow blue lines indicate control signals e.g register file write enable. Although not shown above state elements usually have a reset input to put them into a known state at start-up.

- **Program Counter:** 32-bit register. Its output points to the current instruction, its input indicates the address of the next instruction
- **Instruction Memory:** Has a single read port. It takes a 32-bit instruction address input A, and reads the 32-bit data (i.e instruction) from that address onto the read data output, RD
- **Register File:** 32-element x 32-bit (contains all the defined register $\$s, \$t$ etc. It has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2 specifying one of 32 registers as source operands. They read the 32-bit register values onto read data outputs RD1 and RD2. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

If the address changes, the new data appears at RD after some propagation delay; no clock is involved (they are all read combinationally). They are written only on the rising edge of the clock i.e the state of the system, is changed only at the clock edge. The address,data and write enable must setup sometime before the clock edge and must remain stable until a hold time after the clock edge. Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is built of clocked state elements and combinational logic hence it is also a synchronous sequential circuit, hence the processor can be viewed as an FSM

MIPS Microarchitecture: We consider thre microarchitectures:

- **Single-cycle microarchitecture:** Executes an entire instruction in one cycle. Because it completes the operation in one cycle it does not require any non-architectureal state. However the cycle time is limited by the slowest instruction.
- **Multi-cycle microarchitecture:** Executes instructions in a series of shorter cycles. Simpler instruction mean execution in fewer cycles. It reduces hardware cost by reusing expensive hardware blocks such as adders and memories. It executes only one instruction at a time, but each instruction takes multiple clock cycles.
- **Pipelined microarchitecture:** Applies pipelining to the single-cycle microarchitecture, hence it can execute seeral instructions simultaneously, improving the throughput. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers.

7.2 Performance Analysis

Ways to measure performance:

- The computer that executes your program fastest has the highest performance
- Measure the total execution time of a collection of programs that are similar to those you plan to run (socalled benchmarks) the execution times of these programs are commonly published to give an indication of how a processor performs.

The execution time of a program measured in seconds is given by:

$$Execution\ Time = (\#instructions)\left(\frac{cycles}{instructions}\right)\left(\frac{seconds}{cycle}\right)$$

The number of instructions in a program depends on the porcessor architecture (how complex are the instructions) and on the cleverness of the programmer.

CPI: Cycles per Instruction is the number of clock cycles required to execute an average instruction. (reciprocal of the throughput)

IPC: Instructions per cycle i.e throughput

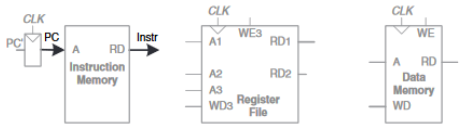
Clock Period: T_c is the number of seconds per cycle. The clock period is determined by the critical path through the logic on the processor.

7.3 Single Cycle Processor

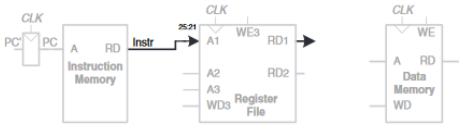
We begin constructing the datapath by connecting the state elements (discussed above) with combinational logic that can execute the various instructions.

Single-Cycle Datapath:

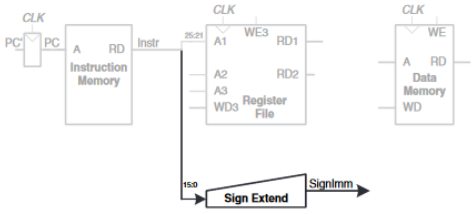
1. The PC register contains the address of the instruction to execute. We read this instruction from instruction memory i.e we connect the PC to the input of the instruction memory. The instruction memory reads out (fetches) the 32-bit instruction (labeled Instr.). The next step depends on the specific instruction that was fetched.



2. assume we fetched a "lw" instruction. We need to read the source register containing the base address which is specified in the rs field of the instruction ($Instr_{25:21}$) These bits are connected to the address input of one of the register file read ports. The register file reads the register value onto RD1.



lw also requires an offset, which is stored in the immediate field of the instruction ($Instr_{15:0}$). Since the 16-bit immediate can either be positive or negative, it must be sign-extended to 32-bits (SignImm).



The processor must add the base address to the offset to find the address to read from memory. We use an ALU to perform this addition. The 3-bit ALUControl signal specifies the operation. The ALU generates a 32-bit ALUResult and a Zero flag which indicates whether the result is 0. ALUResult is sent to the data memory as the address for the load insturction

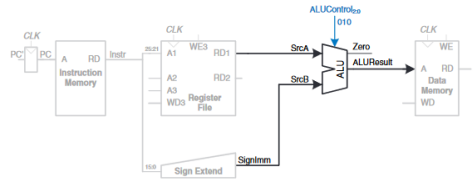


Figure 7.5 Compute memory address

The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle. (Here Port 3 of the reg. file is the write port) The destination register for the "lw" instruction is specified in the rt field ($Instr_{20:16}$) which is connected to the port 3 address input of the register file. The ReadData bus is connected to the port 3 write data input (WD3) of the register file. A control signal "RegWrite" is connected to the port 3 write enable input (WE3) and is asserted during a lw instruction so that the data value is written into the register file.

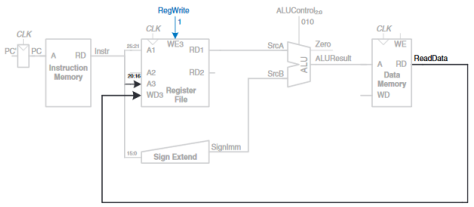
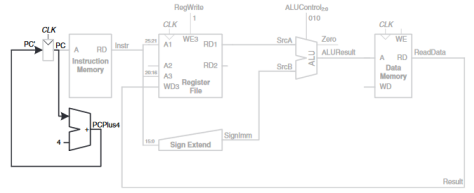
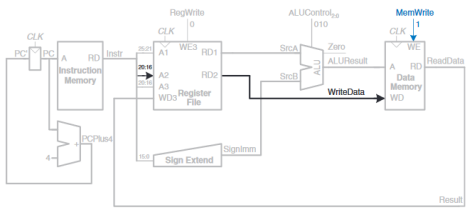


Figure 7.6 Write data back to register file

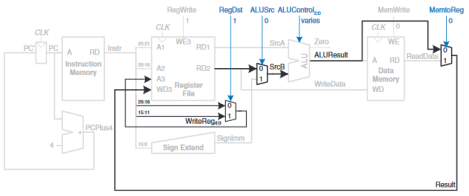
While the instruction is being executed, the processor must compute the address of the next instruction, PC' Because instructions are 32bits = 4 bytes, the next instruction is at PC+4. We use another adder to increment the PC by 4. The new address is written into the Program counter on the next rising edge of the clock. The Datapath for the lw is now complete.



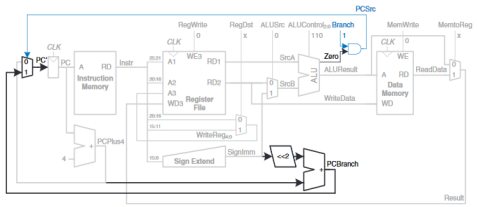
3. We extend the datapath to also handle the "sw" instruction. In addition to the current Datapath the sw instruction also reads a second register from the register file and writes it to the data memory. The register is specified in the rt field $Instr_{20:16}$. These bits of the the instruction are connected to the second register file read port A2. The register value is read onto the RD2 port. It is connected to the write data port of the data memory. The write enable port of the data memory, WE, is controlled by "MemWrite". For a sw instruction MemWrite = 1 to write the data to memory. Data is still read from the address given to the data memory, but ReadData is ignored because RegWrite = 0.



4. Extend the datapath to handle the R-type instructions: add,sub, and,or, and slt. All of these instructions read two registers from the register file, perform some ALU operation on them and write the result back to a third register file. They differ only in the specific ALU operation. Hence they can all be handled with the same hardware, using different ALUControl signals. The register file reads two registers and the ALU performs an operation on these two registers. Instead of taking the SrcB operand from the sign-extended immediate we add a mux to choose SrcB from either the register file RD2 port or SignImm. R-type instructions write the ALUResult to the register file, hence we add another mux to choose between ReadData and ALUResult (we call the output "Result"). This mux is controlled by the signal "MemtoReg", which is 0 for R-type instructions to choose Result from the ALUResult. For R-type instructions the register to write is specified by the "rd" field ($Instr_{15:11}$) hence we add a third mux to choose "WriteReg" from the appropriate field of the instruction. This mux is controlled by "RegDst", which is 1 for R-type instructions.



5. We extend the datapath to handle "beq". beq compares two registers, if they are equal, it takes the branch by adding the branch offset to the PC. The offset is a pos or neg number, stored in the imm field of the instruction ($Instr_{15:0}$). The offset indicates the number of instructions to branch past, hence the immediate must be sign-extended and multiplied by 4 to get the new program counter value ($PC' = PC + 4 + SignImm \cdot 4$). The next PC value for a taken branch, "PCBranch", is computed by shifting SignImm left by 2 bits, then adding it to "PCPlus4". The two registers are compared by computing SrcA-SrcB using the ALU. If ALUResult is 0 (indicated by the Zero flag) the registers are equal. We use a mux to choose PC' from either PCPlus4 or PCBranch. PCBranch is selected if the instruction is a branch and the Zero flag is asserted. (the Signal Branch is 1 for beq and 0 for other instructions.)



Single-Cycle Control: The control unit computes the control signals based on the opcode ($Instr_{31:26}$) and funct ($Instr_{5:0}$) fields of the instruction.

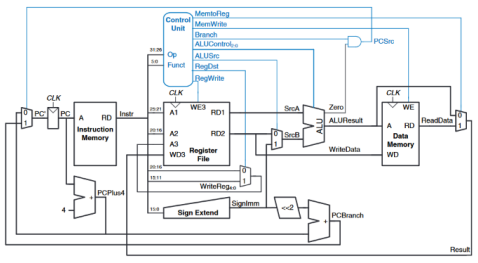
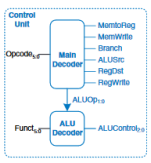


Figure 7.11 Complete single-cycle MIPS processor

R-type instructions also use the funct field to determine the ALU operation. For this reason we factor the control unit into two blocks of combinational logic. The main decoder computes most of the outputs from the opcode, aswell as a 2-bit ALUOp signal. The ALU decoder uses ALUOp signal in conjunction with the funct field to compute ALUControl

Table 7.1 *ALUOp* encoding

ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a



Because ALUOp is never 11, the truth table for the ALU decoder can use dont cares X1 and 1X instead of 01 and 10 to simplify the logic. Note that, for R-type instructions we implement the first two bits of the funct field are always 10, so we may ignore them to simplify the decoder. For the main decoder all R-type instructions use the same main decoder values; they differ only in the ALU decoder output. For instructions that dont write to the register file (e.g sw, beq) we can use dont cares for RegDst and MemtoReg because RegWrite is not asserted.

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Table 7.3 Main decoder truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Supporting more instructions in some cases can be as easy as just enhancing the main decoder, other instructions might also require more hardware in the data path. E.g assume we want to add the "addi" instructions. The datapath is already capable of this task, hence we need to make changes to the controller. We add a new row to the main decoder truth table. We know the opcode for this instruction. The result should be written to the register file hence RegWrite = 1. The destination register is specified in the rt field of the instruction so RegDst = 0. SrcB comes from the immediate, so ALUSrc =1. The instruction is not a branch, nor does it write memory, so Branch=MemWrite = 0. The result comes from the ALU, not memory so MemtoReg = 0. Finally the ALU should add, so ALUOP = 00.
Another example is the "j" instruction. The jump instuction writes a new value into the PC. The two lsb's of the PC are always 0, because the PC is word aligned. The next 26 bits are taken from the jump addres field in $Instr_{25:0}$. The upper four bits are taken from the old value of the PC. The existing datapath lacks the hardware to compute PC', hence we start by adding the necessary hardware to compute PC' when a j instruction is given aswell as a mux to select this next PC. The new mux uses the control signal "Jump". We now add a row to the main decoder truth table and a column for the Jump signal which is 1 for the j instruction and 0 for all the others. j does not write the register file or memory so RegWrite = MemWrite = 0. Hence we dont care about the computation done in the datapath (i.e all other columns are dont cares.)

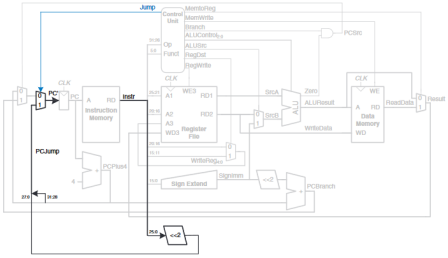


Figure 7.14 Single-cycle MIPS datapath enhanced to support the j instruction

Table 7.5 Main decoder truth table enhanced to support j

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Performance Analysis: Each instruction in a single-cycle processor takes one clock cycle, so CPI = 1. Because we are disciplining ourselves to synchronous sequential design, the clock period is constant and must be long enough to accommodate the slowest instruction. In our current implementation teh lw instructions a has the longest path:

- PC loads a new address on the rising edges of the clock
- Instruction memory reads next instruction
- Register file reads SrcA and immediate field is sign extended and selected at the ALUSrc mux to determinge SrcB
- ALU adds SrcA and SrcB to find the effective address
- Data memory reads from this address
- MemtoReg mux selects ReadData
- Result must setup at the register file before the next rising clock edge

$\Rightarrow T_c = t_{pcq\_PC} + t_{mem} + max[t_{RFread}, t_{sext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
The critical path is shown below.

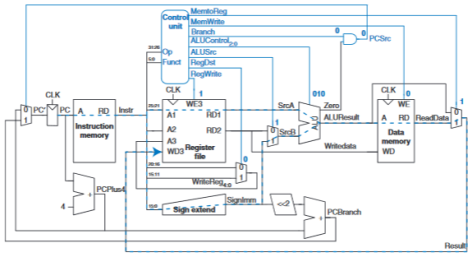


Figure 7.15 Critical path for `lw` instruction

7.4 Multicycle Processor

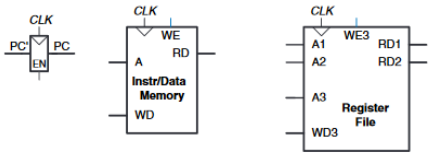
The single-cycle processor has 3 primary weaknesses:

1. It requires a clock cycle long enough to support the slowest instruction eventhough most instructions are faster
2. It requires 3 adders, which are relatively expensive circuits (especially if they must be fast)
3. It has seperate instruction and data memories which may not be realistic. (Most computers have a single large memory that holds both instructions and data and that can be read and written)

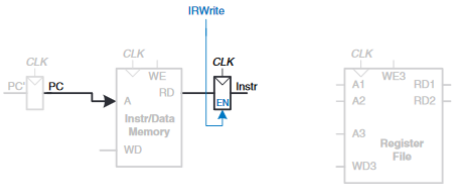
The multicycle processor addresses these problems by breaking an instruction into multiple shorter steps. In each step teh processor can read or write the memory or register file or use the ALU. Different instructions use different number of steps, hence simpler instructions can complete faster than more complex ones. The processor only uses one adder which is reused for different purposes on various steps. It also uses combined memory. The instruction is fetched from memory from memory on the first step, and data may be read or written on later steps. In contrast to the design of the single cycle processor we also add nonarchitectural state elements to hold intermediate results between the steps. When designing the controller we must take into account that it produces different signals on different steps during the execution of a single instruction hence it is and FMS instead of combinational logic.

Multicycle Datapath:

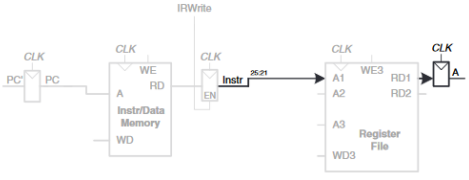
1. Instead of having seperate Instruction and Data Memory, we unify them so that we can read the instruction in one cycle, then read/write data in a seperate cycle. The PC and register gile remain the same as in the single cycle processor.



2. The PC contains the address of the instruction to execute. We first read this instruction from instruction memory. We do this by connecting PC to the address input of the instruction memory. The instruction is read and stored in a new nonarchitectural Instruction Register so that it is available for future cycles. The instruction Register receives an enable signal "IRWrite" that is asserted when it should be updated with a new instruction.



3. We will work out the data-path connections for the lw instruction then enhance it to handle the others. For lw the next step is to read the source register containing the base address, which is specified in the rs field of the instruction ($Instr_{25:21}$) These bits are connected to one of the Address inputs, A1, of the register file. The register file reads the register onto RD1. This value is stored in another nonarchitectural register A.



lw also requires an offset, which is stored in the immediate field of the instruction ($Instr_{15:0}$) and must be sign-extended to 32-bits. The sign extended value is called "SignImm". We do not need a nonarchitectural register for SignImm because it is a combinational function of Instr and will not change while the current instruction is being processed.

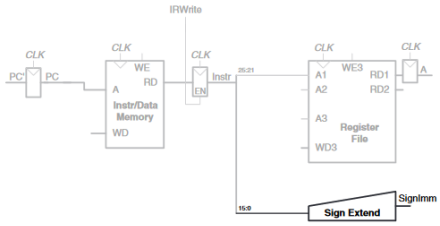


Figure 7.19 Sign-extend the immediate

The address of the load is the sum of the base address and the offset. We use an ALU to compute this sum. The ALUResult is stored in a nonarchitectural register called ALUOut.

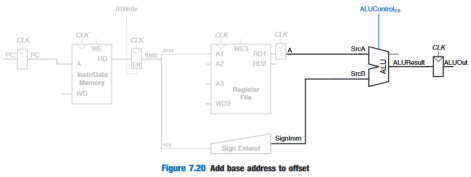


Figure 7.20 Add base address to offset

The next step is to load the data from the calculated address in the memory. We add a mux in front of memory to choose the memory address, "Adr", from either the PC or ALUOut. The mux select signal is called IorD, to indicate either an instruction or data address. The data read from the memory is stored in another nonarchitectural register, called "Data". (This mux allows us to reuse the memory during the lw instruction, hence IorD will have different values on different steps of the instruction)

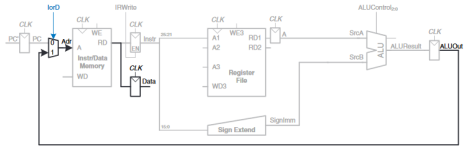


Figure 7.21 Load data from memory

Finally, the data is written back to the register file. The destination register is specified by the rt field of the instruction ($Instr_{20:16}$)

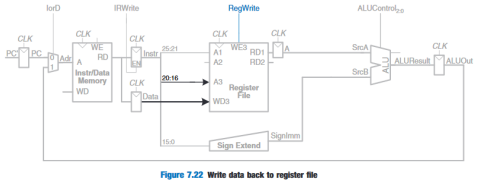


Figure 7.22 Write data back to register file

While this is happening, the processor must update the program counter by adding 4 to the old PC. In the multicycle processor, we can use the existing ALU on one of the steps when it is not busy. In order to do this, we must insert source mux0s to choose the PC and the constant 4 as ALU inputs. A two-input mux controlled by ALUSrcA chooses either the PC or register A as SrcA. A four-input mux controlled by ALUSrcB chooses either 4 or SignImm as SrcB (other inputs will be used later when enhancing the datapath for other instructions). The PC is updated when the ALU adds SrcA to SrcB and the result is written into the program counter register. "PCWrite" control signal enables the PC register to be written only on certain cycles.

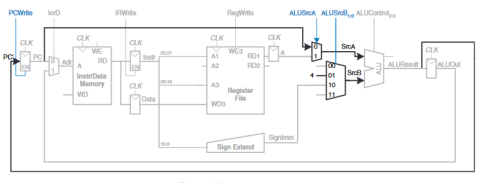


Figure 7.23 Increment PC by 4

4. We extend the datapath to also handle the sw instruction. The sw instruction reads a base address from port 1 of the register file and sign-extends the immediate. The ALU adds the base address to the immediate to find the memory address. These functions are already supported by the existing hardware in the datapath. The new feature of sw is, that we must read a second register from the register file and write it into the memory. The register is specified in the rt field of the instruction ($Instr_{20:16}$) which is connected to teh second port of the register file. When the register is read, it is stored in a nonarchitectural register, B. In the next step it is sent to the write data port (WD) of the data memory to be written. The memory receives an additional MemWrite control signal to indicate that the write should occur.

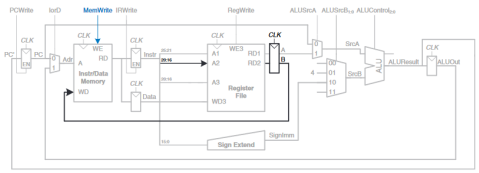


Figure 7.24 Enhanced datapath for `sw` instruction

5. For R-type instructions we again fetch the instruction and read two source registers from the register file. We choose register B as the second source register for the ALU. The ALU performs the appropriate operation and stores the result in ALUOut. On the next step, ALUOut is written back to the register specified by the rd field of the instruction ($Intrs_{15:11}$). This requires two new mux's. The MemtoReg mux selescts whether WD* comes from ALUOut (for R-type instructions) or from Data (for lw). The RegDst instruction selects whether the destination register is specified in the rt or rd field of the instruction.

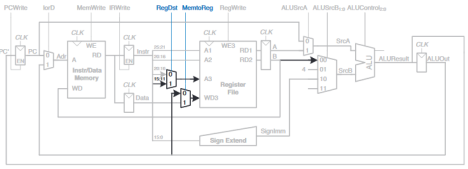


Figure 7.25 Enhanced datapath for R-type instructions

6. For the beq instruction, we again fetch the instruction and read the two source registers from the register file. To determine whether the registers are equal the ALU subtracts the registers and, upon a zero result sets the Zero flag. Meanwhile the datapath must compute the next value of the PC if the branch is taken: $PC' = PC + 4 + SignImm \cdot 4$. In the Multicycle processor, the ALU is reused to save hardware. On one step, the ALU computes PC+4 and writes it back to the program counter, as was done for other instructions. On another step, the ALU uses this updated PC value to compute PC'. SignImm is left-shifted by 2 to multiply it by 4. The SrcB mux chooses this value and adds it to the PC. This sum is the destination of the branch and is stored in ALUOut. A new mux controlled by PCSrc, chooses what signal should be sent to PC'. The program counter should be written when either PCWrite is asserted or when a branch is taken. The control signal "Branch", indicates that the beq instruction is being executed. The branch is taken if Zero is also asserted. Hence, the datapath computes a new PC write enable "PCEn", which is TRUE when either PCWrite is asserted or when both Branch and Zero are asserted.

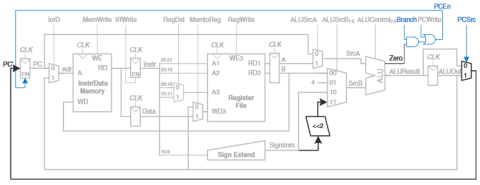


Figure 7.26 Enhanced datapath for `beq` instruction

Multicycle Control: The control unit computes the control singals just as the single-cycle processor (i.e form opcode and funct fields)

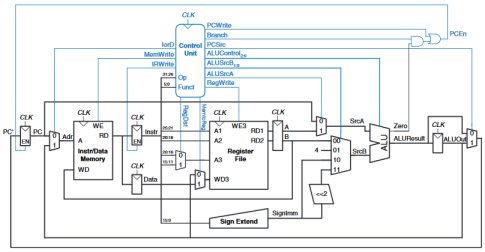


Figure 7.27 Complete multicycle MIPS processor

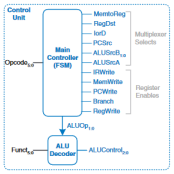


Figure 7.28 Control unit internal structure

As in the single-cycle processor the control unit is partitioned into a main controller and an ALU decoder. The ALU decoder is unchanged. However the main controller is an FSM that applies the proper control signals on the proper cycles or steps. The sequence of control signals depends on teh instruction being executed. Our goal is to create the FSM state transition diagram for the main controller:
The main controller produces mux select and register enable signals for the datapath. For the state transition diagrams only relevant control signals are listed. Select signals are listed only when their value matters; otherwise they are dont cares. Enable signals are listed only when they are asserted; otherwise they are 0.

1. Fetch the instruction from memory at the address held in the PC. The FSM enters this state on reset.
   - IorD = 0 so the address is taken from the PC
   - IRWrite = 1 to write the instruction into the instruction register.
   - Because the ALU is not being used for anything, the processor can use it to compute PC+4 as it fetches the instruction.
   - ALUSrcA = 0, so SrcA comes from the PC.
   - ALUSrcB = 01, so SrcB is the constant 4.
   - ALUOp = 00, so the ALU decoder produces ALUControl = 010 to make the ALU add.
   - PCSrc = 0, PCWrite = 1 to update the PC with the new value.

Figure 7.29 Fetch

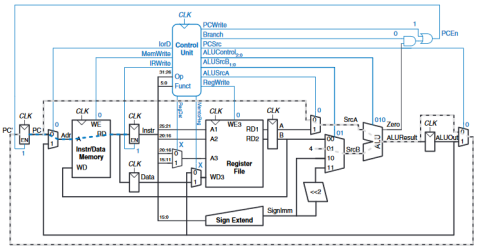


Figure 7.30 Data flow during the fetch step

(The instruction fetch shown by dashed blue line, PC increment by dashed gray line)

2. Read the register file and decode the instruction. The register file always reads the two sources specified by the rs and rt fields of the instruction. Meanwhile the immediate is sing-extended. Decoding involves examining the opcode of the instruction to determine whats next. No control singals are necessary to decode the instruction, but the FSM must wait 1 cycle for the reading and decoding to complete.

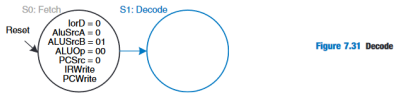


Figure 7.31 Decode

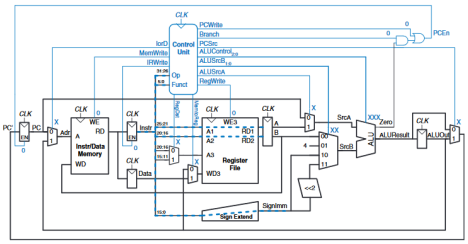
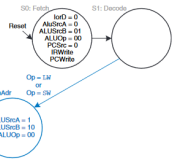


Figure 7.32 Data flow during the decode step

3. The FSM proceeds to one of several possible states, depending on the opcode.
   - If the instruction is lw or sw the multicycle processor computes the address by adding the base address to the sign-extended immediate. This requires:
     - ALUSrcA = 1 to select register A
     - ALUSrcB = 10 to select SignImm
     - ALUOp = 00, so the ALU adds

     The effective address is stored in the ALUOut register for use on the next step.

Figure 7.33 Memory address computation



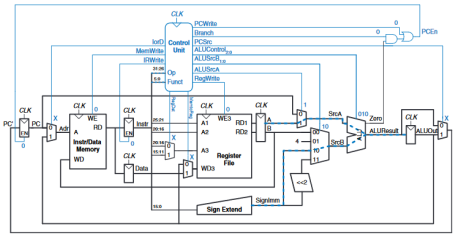
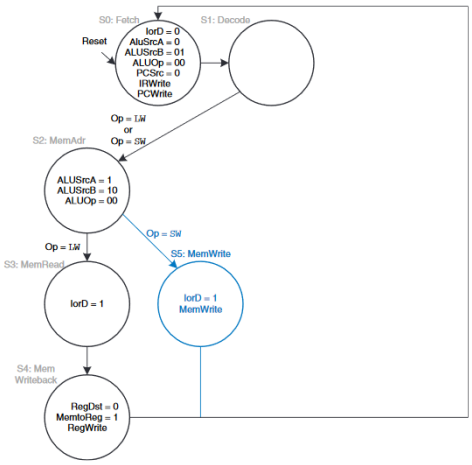


Figure 7.34 Data flow during memory address computation

   - If the instruction is lw, the processor must next read data from memory and write it to the register file:
     - IorD =1 to select the memory address that was saved in ALUOut. This is read and saved in the Data register during step S3.
     - MemtoReg = 1 to select "Data" (Step S4)
     - RegDst = 0 to pull destination register from the rt field of the instruction (S4)
     - RegWrite = 1 to perform the write.

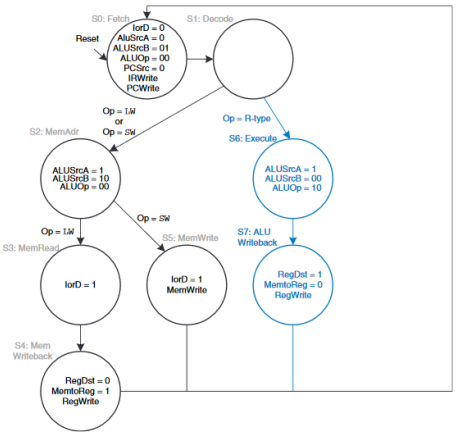
     The FSM returns to the initial state, S0, to fetch the next instruction.
   - If the instruction is sw, the data read from the second port of the register file is simply written to memory:
     - IorD = 1 to select the address in ALUOut
     - MemWrite = 1 to write to memory

     The FSM returns to S0 to fetch the next instruction



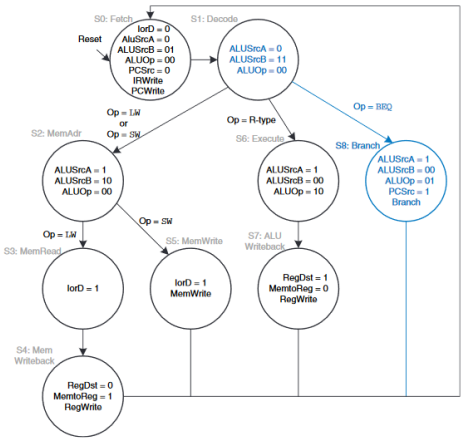
- If the opcode indicates an R-type instruction, the processor must calculate the result using the ALU and write that result to the register file. In S6:
  - ALUSrcA = 1, ALUSrcB= 00 to select the registers A and B
  - ALUOp = 10 for all R-type instructions.
  - ALUResult is stored in ALUOut.

  In S7:
  - RegDst = 1, because the destination register is specified in the rd field of the instruction.
  - MemtoReg = 0 because the write data, WD3, comes from ALUOut.
  - RegWrite is asserted to write the register file.

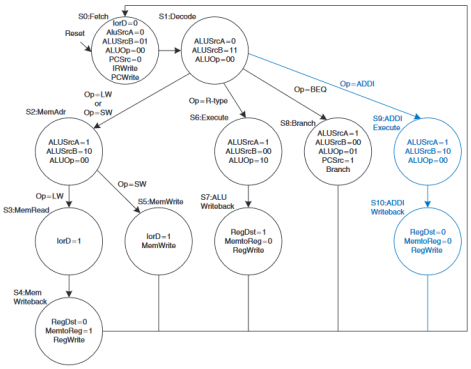


- For a beq instruction the processor must calculate the destination address and compare the two source registers to determine whether the branch should be taken. The ALU was not used during S1, when the registers were being read. The processor can use teh ALU at that time to compute the destination address:
  - ALUSrcA = 0 to select the incremented PC
  - ALUSrcB = 11 to select $SignImm \cdot 4$
  - ALUOp = 00 to add

  The destination address is stored in ALUOut. If the instruction is not beq the computed address will not be used in subsequent cycles.
  In S8 the processor compares the two registers by checking if their difference is 0. If it is the processor branches to the address that was just computed:
  - ALUSrcA = 1 to select Register A; ALUSrcB = 00 to select Register B
  - ALUOp = 01 to subtract
  - PCSrc = 1 to take the destination address from ALUOut
  - Branch = 1 to update the PC with this address if the ALU result is 0



More Instructions: We extend the processor to support the addi and j instructions.
addi: The datapath is already capable of adding registers to immediates, so we only need to add new states to the main controller FSM. In S9, register A is added to SignImm and the result ALUResult is stored in ALUOut. In S10 ALUOut is written to the register specified by the rt field of the instruction



j: First we must modify the datapath to compute the next PC value in the case of a j instruction. We then add a state to the main controller to handle the instruction. The jump destination is formed by left-shifting the 26-bit addr field of the instruction by two bits, then prepending the four msb's of tehe already incremented PC. PCSrc mux is extended to take this address as a third input. The new state S11, selects PC' as the PCJump value and writes the PC (the PCSrc signal is extended to two bits in S0 and S8 as well)

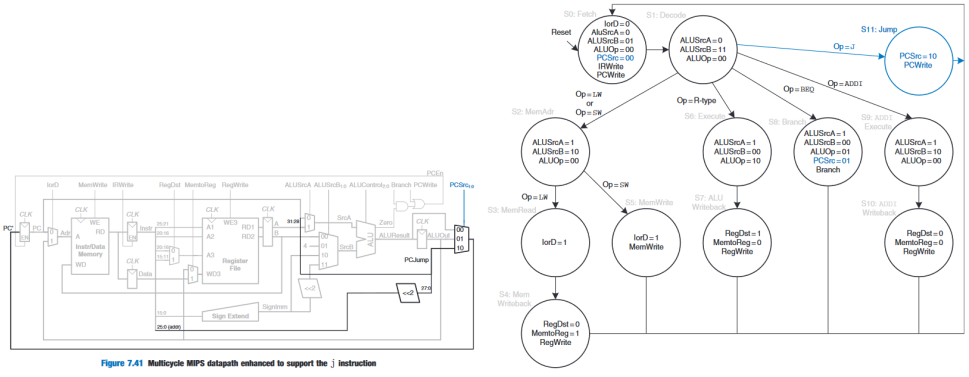


Figure 7.41 Multicycle MIPS datapath enhanced to support the j instruction

Performance Analysis: The execution time of an instruction depends on both the number of cycles it uses and the cycle time. The multicycle processor does less work in a single cycle compared to the single cycle processor, hence it also has a shorter cycle time. It requires 3 cycles for beq and j instructions, 4 cycles for sw addi and R-type instructions and 5 cycles for lw instructions. The CPI depends on the relative likelihood that each instruction is used. The processor was designed such that each cycle involved one ALU operation, memory access or register file access. We assume writing is faster than reading memory and that the register file is faster than memory. We have two critical paths that would limit the cycle time:

$$T_c = t_{pcq} + t_{mux} + max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

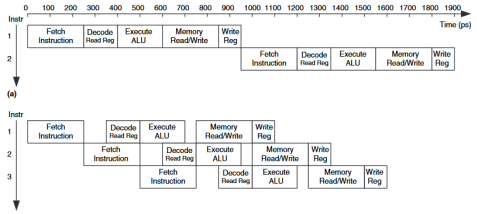
Multicycle processors can be slower than single-cycle processors in certain cases. The problem is that even though the slowest instruction, is broken into more steps, the multicycle processor cycle time might not improve by that factor. This is partly because not all of the steps are exactly the same length, and partly because sequencing overhead of the register clk-to-Q and setup time must now be paid on ecery step, not just once for the entire instruction. The multicycle processor is likely to be less expensive becasue it eliminates two adders and combines the instruction and data memories into a singe unit.

7.5 Pipelined Processor

We desing a pipelined processor by subdividing the single-cycle processor into five pipeline stages i.e 5 instructions can be executed simultaneously, one in each stage. Because each stage only has one-fifth of the entire logic, the clock frequency is almost 5 times faster. Hence the latency of each instruction is ideally unchanged, but the throughput is ideally 5 times better.
Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose 5 pipeline stages so that each stage involves exactly one of these slow steps. The 5 stages are called:

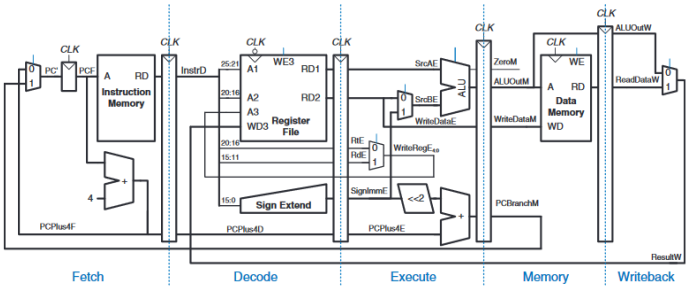
- Fetch: the processor reads the instruction from instruction memory
- Decode: the processor reads the source operands from the register file and decodes the instruction to produce the control signals.
- Execute: The processor performs a computation with the ALU
- Memory: The processor reads or writes data memory
- Writeback: The processor writes the result to the register file when applicable

Below is a comparison between a single-cycle and a pipelined processor.



In a pipelined processor, the register file is written in the first part of a cycle and read in the second part. (so that data can be written and read back within a single cycle. The challenge for pipelined systems is handling hazards that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed

Pipelined Datapath: The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers.



The Register file is special because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage but the write address and data come from the Writeback stage. This feedback leads to pipeline hazards. The register file in the pipelined processor writes on the falling edge of CLK, when WD3 is stable. All signals associated with a particular instruction must advance through the pipeline in unison. In the above figure there is an error related to this issue:
The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from ResultW, a Writeback stage signal, but the address comes from WriteRegE, and Execute stage signal. By sending the signal through the Memory and Writeback stages we can sync it with the rest of the instruction.

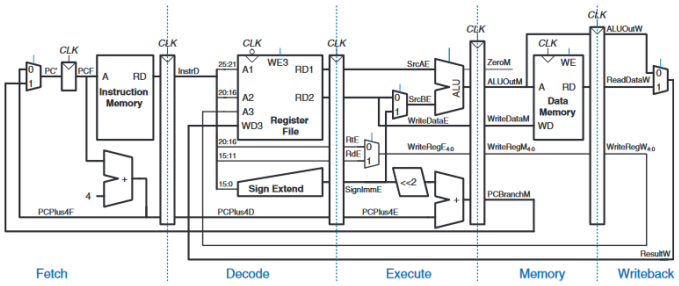


Figure 7.46 Corrected pipelined datapath

Pipelined Control: The pipelined processor takes the same control signals as teh single-cycle processor and therefore uses the same control unit. It examines the opcode and funct fields of the instruction in the Decode stage to produce the control signals. these control signals must be pipelined along with the data so that they remain synchronized with the instruction. RegWrite must be pipelined into the Writeback stage before it feeds back to the register file, just as WriteReg was pipelined.

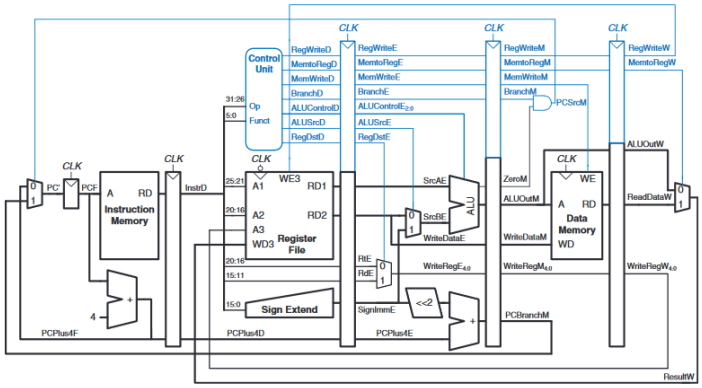


Figure 7.47 Pipelined processor with control

Hazards: In a pipelined system, multiple instructions are handled concurrently. When one instruction is dependent on the results of another that has not yet completed, a hazard occurs. The register file can be read and written in the same cycle. The write takes place during the first half of the cycle and the read during the second half, so a register can be written and read back in the same cycle without introducing a hazard.
Read after write (RAW) hazard: This hazard occurs when one instruction writes a register and subsequent instructions read this register.

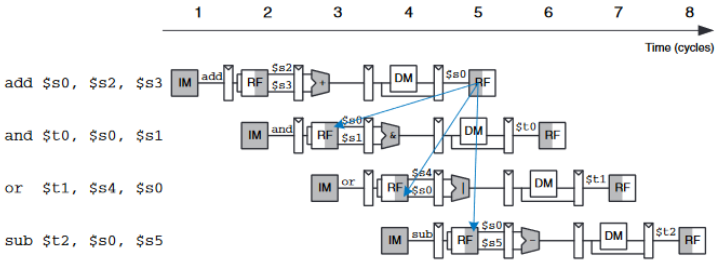


Figure 7.48 Abstract pipeline diagram illustrating hazards

The add instruction writes a result into $\$s0$ in the first half of cycle 5. However, the and instruction reads $\$s0$ on cycle 3, obtaining the wrong value, the same for the or instruction. The sub instruction obtains the correct value aswell as andy subsequent instructions.
Hazards are classified as data hazards or control hazards:

- data hazard: occurs when an instruction tries to read a register that has not yet been written back by a previous instruction.
- control hazard: occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

Solving data hazards with forwarding (bypassing): Some data hazards can be solved by forwarding a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding mux's in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Example:

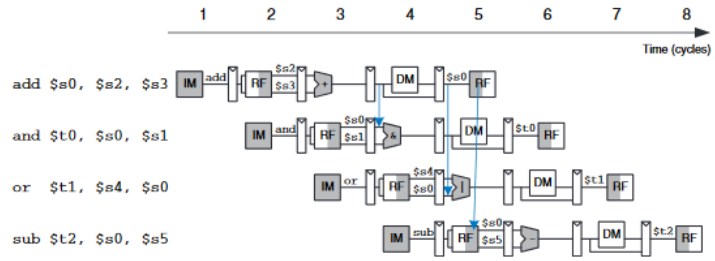


Figure 7.49 Abstract pipeline diagram illustrating forwarding

In cycle 4 s0 is forwarded from the Memory stage of the add instruction to thze Execute stage of the dependent and instruction.
Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instrcution in the Memory or Writeback stage.
We can modify the processor to support forwarding, by adding a hazard detection unit and two forwarding mux's. The hazard detection unit receives the two source registers from the instruction in the Execute stage and the destination registers from the intructions in the Memory and Writeback stages. It also recieves the RegWrite signals from the Memory and Writeback stages to know whether the destination register will actually be written.(e.g sw, beq do not write results to register file hence their results dont need to be forwarded.)

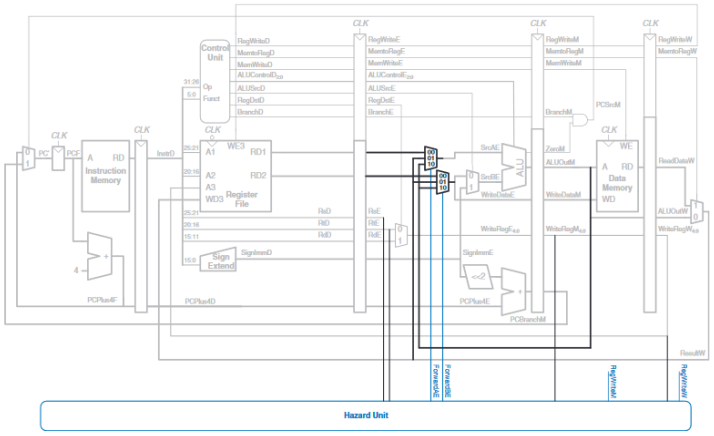


Figure 7.50 Pipelined processor with forwarding to solve hazards

The hazard detection unit computes control signals for the forwarding mux's to choose operands from the register file or from the results in the Memory or Writeback stage. $0 is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, them Memory stage should have priority because it contains the more recently executed instruction.

Solving Data Hazards with Stalls: For cases like the lw instruction where reading the data only finishes at the end of the Memory stage, results cannot be forwarded to the Execute stage of the next instruction. lw has a **two-cycle latency** i.e a dependent instruction cannot use its result until two cycles later. The figure below illustrates this problem.

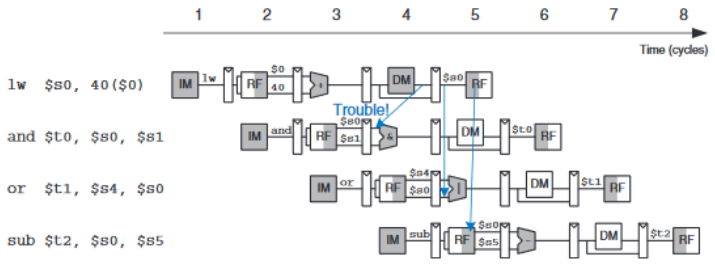


Figure 7.51 Abstract pipeline diagram illustrating trouble forwarding from lw

The alternative solution is to stall the pipeline i.e holding up operation until the data is available.

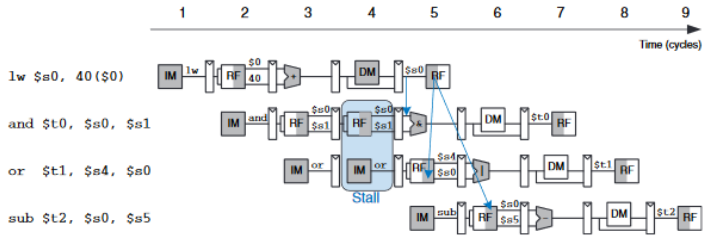


Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

The or must remain in the Fetch stage during both cycles as well, becasue the Decode stage is full. From cycle 4 there is unused stage propagation through the pipeline, this is called a **bubble** (behaves like a nop instruction).
Stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward. Stalls degrade performance and should only be used when necessary. We modify teh pipelined processor to add stalls for lw data dependencies.
The hazard unit examines the instruction in the Execute stage, if its lw and its destination register matches either source operand of the instruction in the Decode stage, that instruction must be stalled in the Decode stage until the source operand is ready. Stalls are done by adding enable inputs to the Fetch and Decode pipeline registers and a synchronous reset/clear input to the Execute pipeline register. When a stall occurs StallD and StallF are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. FlushE is also asserted to clear the contents of the Execute stage pipeline regtister. (introduces bubble.)

Solving Control Hazards: The beq instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. One solution would be to stall the pipeline until the branch decision is made, but since the decision is made in the Memory stage, the pipeline would have to be stalled for three cycles at every branch, degrading performance. An alternative is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor throws out the instruction if the prediction was wrong. In the case the prediction was wrong the 3 instructions following the branch must be flushed. These wasted instruction cycles are called **branch misprediction penalty**. This is an improvement over stalling ,but flushing so many instructions when the branch is taken still degrades performance.

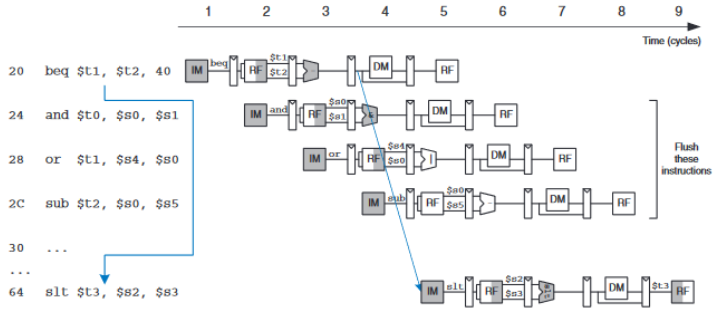


Figure 7.54 Abstract pipeline diagram illustrating flushing when a branch is taken

Branch misprediction penalty can be reduced by using a dedicated equality comparator (Making the decision simply requires comparing the values of two registers). If the comparator is fast enough, it could be moved back into the Decode stage, hence the next PC can be determined by the end of the Decode stage.

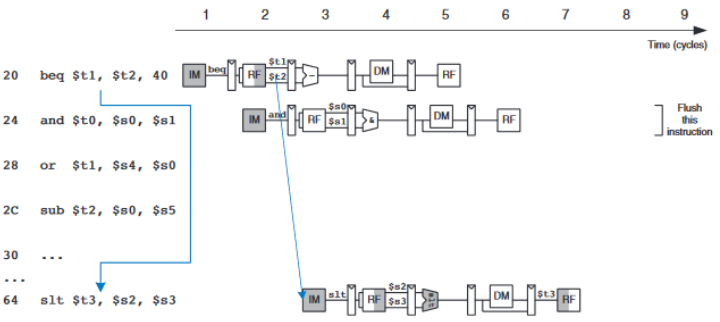


Figure 7.55 Abstract pipeline diagram illustrating earlier branch decision

We have reduced the branch misprediction penalty to one instruction. We modify the processor to move the branch decision earlier and handle control hazards.

- Add equality comparator to the Decode stage

- PCSrc AND gate moved earlier so that PCSrc can be determined in the Decode stage rather than the Memory stage.
- PCBranch adder moved into the Decoded stage such that destination address can be computed in time.
- Synchronous clear input (CLR) connected to PCSrcD is added to the Decode stage pipeline register so that incorrectly fetched instruction can be flushed when a branch is taken.

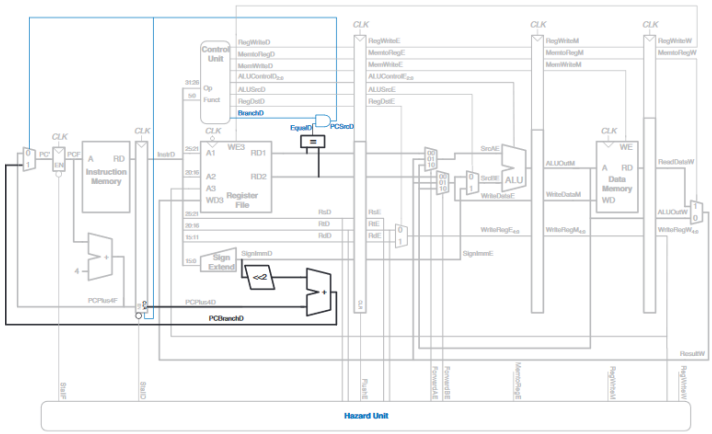


Figure 7.56 Pipelined processor handling branch control hazard

The early branch decision hardware introduces a new RAW data hazard. If one of the source operands for the branch was computed by a previous instruction and has not yet been written into the register file, the branch will read the wrong operand value from the register file. This hazard can be solved by forwarding the correct value if it is available or stalling until data is ready. We make modifications to the processor:
If the result of an ALU instruction is in the Memory stage, it can be forwarded to the equality comparator through two new mux's. If the result of an ALU instruction is in the Execute stage or the result of a lw instruction is in the Memory stage we must stall at the Decode stage until the result is ready.

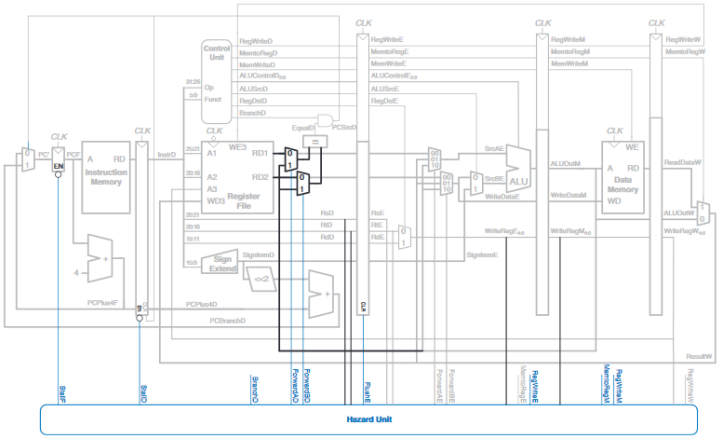


Figure 7.57 Pipelined processor handling data dependencies for branch instructions

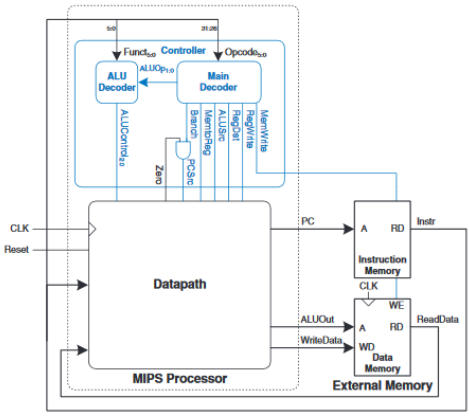
More Instructions: Supporting new instructions in the pipelined processor is like supporting them in the single-cycle processor, new instructions may introduce hazards that must be detected and solved.

Performance Analysis: Ideally it would have a CPI of 1, because a new instruction is issued every cycle. A stall or a flush wastes a cycle, so the CPI is higher and depends on the specific program being executed. Cycle time can be determined by considereing the critical path in each of the five pipelining stages. The register file is written in the first half of the WRiteback cycle and read in the second half of the Decode cycle, hence the cycle time of the Decode and Writeback stage is twice the time necessary to do the half-cycle of work.

$$T_c = max \left(\begin{array}{c} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + T_{mux} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{array} \right) \begin{array}{c} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{array}$$

7.6 HDL Representation

Single-Cycle Processor: The Following is MIPS single-cycle processor interfaced to external memory aswell as the Systemverilog for each component.



HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

HDL Example 7.4 ALU DECODER

SystemVerilog

```
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110; // sub (for beq)
      default: case(funct)         // R-type instructions
          6'b100000: alucontrol <= 3'b010; // add
          6'b100010: alucontrol <= 3'b110; // sub
          6'b100100: alucontrol <= 3'b000; // and
          6'b100101: alucontrol <= 3'b001; // or
          6'b101010: alucontrol <= 3'b111; // slt
          default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule
```

SystemVerilog

```
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata);

  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero;
  logic [2:0] alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);
endmodule
```

HDL Example 7.3 MAIN DECODER

SystemVerilog

```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic         clk, reset,
                input  logic         memtoreg, pcsrc,
                input  logic         alusrc, regdst,
                input  logic         regwrite, jump,
                input  logic [2:0]   alucontrol,
                output logic         zero,
                output logic [31:0]  pc,
                input  logic [31:0]  instr,
                output logic [31:0]  aluout, writedata,
                input  logic [31:0]  readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;

  // next PC   logic
  flopr #(32)  pcreg(clk, reset, pcnext, pc);
  adder        pcadd1(pc, 32'b100, pcplus4);
  sl2          immsh(signimm, signimmsh);
  adder        pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)   pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)   pcmux(pcnextbr, {pcplus4[31:28],
                     instr[25:0], 2'b00}, jump, pcnext);

  // register file logic
  regfile    rf(clk, regwrite, instr[25:21], instr[20:16],
                writereg, result, srca, writedata);
  mux2 #(5)  wrmux(instr[20:16], instr[15:11],
                   regdst, writereg);
  mux2 #(32) resmux(aluout, readdata, memtoreg, result);
  signext    se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb, alucontrol, aluout, zero);
endmodule
```

Generic Building Blocks: The following is SystemVerilog for generic building blocks useful in any MIPS microarchitecture.

HDL Example 7.6 REGISTER FILE

SystemVerilog

```
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clk
  // register 0 hardwired to 0
  // note: for pipelined processor, write third port
  // on falling edge of clk

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

HDL Example 7.7 ADDER

SystemVerilog

```
module adder(input  logic [31:0] a, b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

HDL Example 7.8 LEFT SHIFT (MULTIPLY BY 4)

SystemVerilog

```
module sl2(input  logic [31:0] a,
           output logic [31:0] y);
  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
```

HDL Example 7.9 SIGN EXTENSION

SystemVerilog

```
module signext(input  logic [15:0] a,
               output logic [31:0] y);
  assign y = {{16{a[15]}}, a};
endmodule
```

HDL Example 7.10 RESETTABLE FLIP-FLOP

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

HDL Example 7.11 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

Testbench: The MIPS testbench loads a program into the memories. The testbench, top-level MIPS module, and external memory HDL code are given in the following examples:

HDL Example 7.12 MIPS TESTBENCH

SystemVerilog

```
module testbench();

  logic  clk;
  logic  reset;

  logic [31:0] writedata, dataadr;
  logic        memwrite;

  // instantiate device to be tested
  top dut (clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if (memwrite) begin
        if (dataadr === 84 & writedata === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

HDL Example 7.13 MIPS TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic        memwrite);

  logic [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule
```

HDL Example 7.14 MIPS DATA MEMORY

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

HDL Example 7.15 MIPS INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic [5:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat", RAM);

  assign rd = RAM[a]; // word aligned
endmodule
```

7.7 Exceptions

Exceptions cause unplanned changes in the flow of a program. We will enhance the multicycle processor to support two types of exceptions:

- Undefinded instructions
- Arithmetic overflow

When an exception takes place, the processor copies the PC to the EPC register and stores a code in the Cause register indicating the source fo the exception.

- 0x28 for undefined instructions
- 0x30 for overflow

The processor jumps to the exception handler at memory address 0x80000180. It is code that responds to the exception and part of the OS. The exception registers are part of Coprocessor 0, a portion of the MIPS processor that is used for system functions. Coprocessor 0 defines up to 32 special-purpose registers. The exception handler may use the "mfc0" (move from coprocessor 0) instruction to copy these special purpose registers into a general purpose register in the register file. To handle exceptions, EPC and Cause registers must be added to the datapath and extend PCSrc mux to accept the exception handler address.

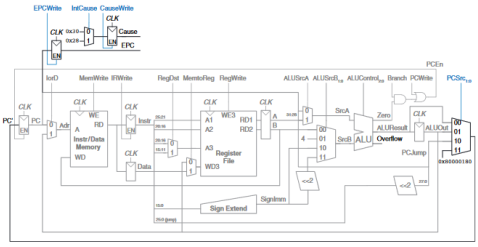


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

The two added registers have write enables to store the PC and exception cause when an exception takes place. The cause is generated by a mux which selects the appropriate code for the exception. The ALU must also generate and over flow signal.
To support the mfc0 instruction, we add a way to select the Coprocessor 0 registers and write them to the register file.

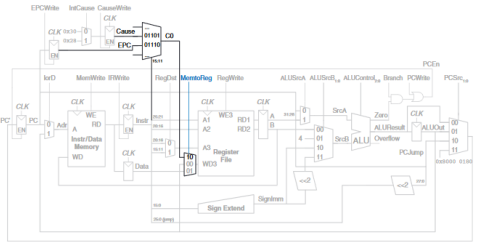


Figure 7.63 Datapath supporting mfc0

We must also modify the controller. The controller recieves the overflow flag from the ALU. It generates 3 new control signals:

- Write the EPC
- Write the Cause register
- Select the Cause

And two new states, one to support the two exceptions and another to handle mfc0. If the controller recieves an undefined instructions it proceeds to S12, saves teh PC in EPC, writes 0x28 to the Cause register, and jumps to the exception handler. If the controller detects arithmetic overflow on an add or sub instruction, it proceeds to S13, saves teh PC in EPC, writes 0x30 in the Cause register and jumps to the exception handler. When an exception occurs the instruction is discarded and the register file is not written. When a mfc0 instruction is decoded, the processor goes to S14 and writes the appropriate Coprocessor 0 register to the main register file.

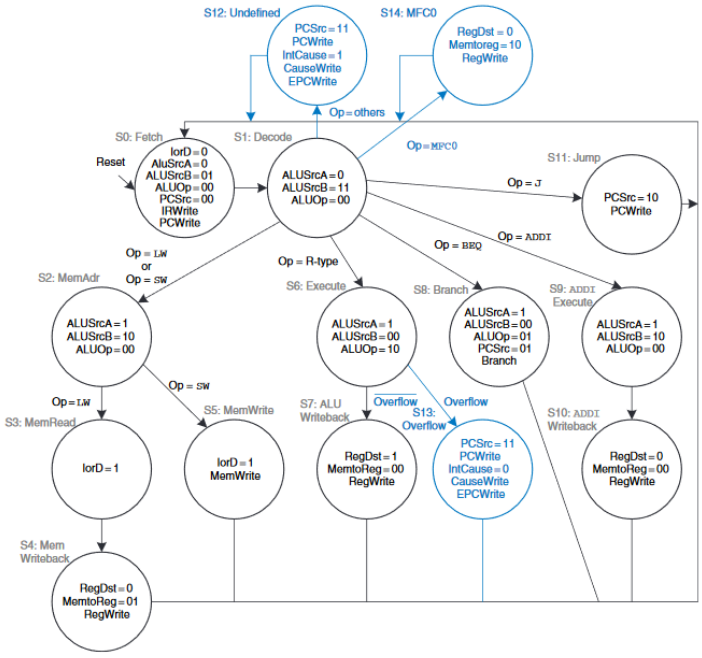


Figure 7.64 Controller supporting exceptions and mfc0

7.8 Advanced Microarchtecture:

The time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction.
⇒ To increase performance we need to speed up the clock and/or reduce the CPI

Deep Pipelines: The easiest way to speed up the clock is to chop the pipeline into more stages. Each stage can run faster because it contains less logic. The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead and cost. Longer pipelines introduce more dependencies. (resolving the dependencies with stalls increases the CPI). The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay. This sequencing overhead makes adding more pipeline stages give diminishing returns. Adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Branch Prediction: To reduce branch misprediction penalties most pipelined porcessors use a branch predictor to guess whether the branch should be taken.

- **static branch prediction:** Because loops usually have a high number of iterations the simplest branch predictor checks the direction of the branch and predicts that backward branches should be taken. It is independent of the history of the program).

- **Dynamic branch predictors:** Use the history of program execution to guess whether a abranch should be taken. They maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table is known as a **branch target buffer**, it includes the destination of the branch and a history of whether the branch was taken.
- **One-bit dynamic branch predictor** Remembers whether the branch was taken the last time and predicts that it will do the smae thing the next time. It mispredicts the first and last branches of a loop.
- **2-bit dynamic branch predictor:** Has 4 states:
  - strongly taken
  - weakly taken
  - weakly not taken
  - strongly not taken.

  When the loop is repeating, it enters the strongly not taken state and predicts that the branch should not be taken next time. THis is correct until the last branch of the loop, which is taken and moves the predictor to the "weakly not taken" state. When the loop is first run again the predictor correctly predicts that the branch should not be taken and reenters the strongly not taken state i.e the 2-bit branch predictor mispredicts only the last branch of a loop.

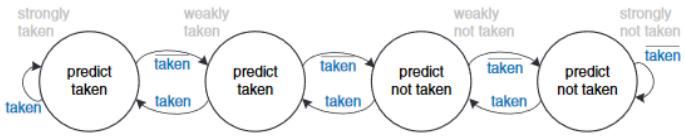


Figure 7.66 2-bit branch predictor state transition diagram

Superscalar Processor: contains multiple copies of the datapath hardware to execute multiple instructions simultaneously.

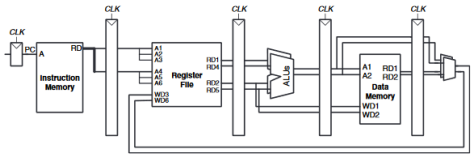


Figure 7.67 Superscalar datapath

The aboce example is a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. Below is the respective pipeline. Executing many instructions simultaneously is difficult because of dependencies

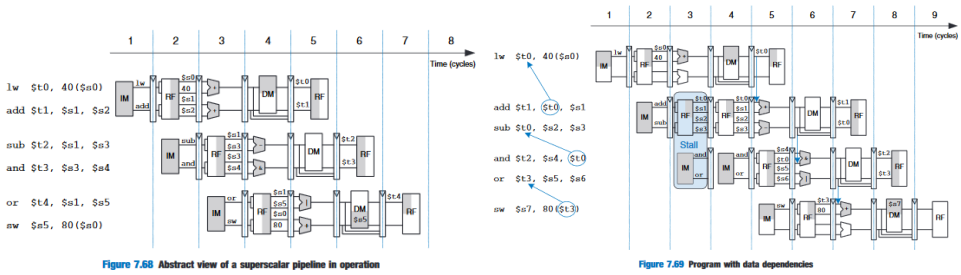
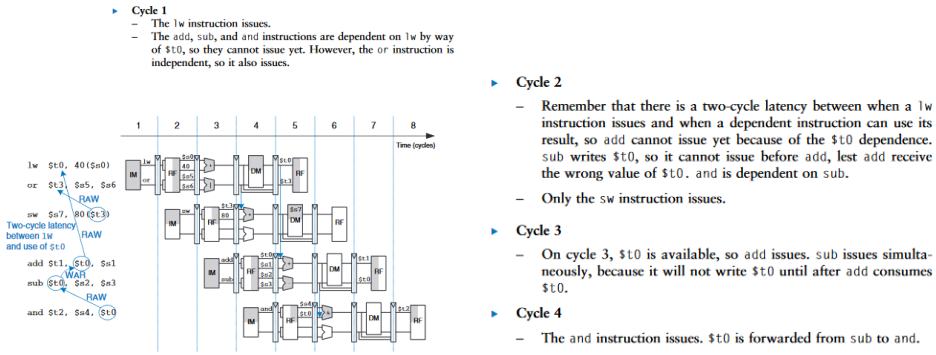


Figure 7.68 Abstract view of a superscalar pipeline in operation

Figure 7.69 Program with data dependencies

Superscalar processors exploit both temporal (pipelining) and spatial parallelism (copies of hardware).

Out-of-Order Processor: Looks ahead across many instructions to issue/begin executing independent instructions as rapidly as possible. (instructions can be issued in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.



The processor issues six instructions in four cycles hence it has an IPC of 1.5. The two hazards in the image above are:

- RAW: Read after Write. An instruction must not read the register until the other instruction has written to it
- WAR: Write after Read (antidependence). An instruction must not write to the register until the other has read from it.WAR hazards could not occur in the simple MIPS pipeline, but may occur in an out-of-order processor. A WAR hazard is not essential to the operation of the program it is the programmers choice to use the same register for two unrelated instructions. (MIPS only has 32 registers i.e sometimes the programmer is forced to reuse a register because all other registers are in use)
- WAW: Write after Write (output dependence). WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being written to the register. WAW hazards are also programmer dependent

Out-of-order processors use a table ("scoreboard") to keep track of instructions waiting to issue. It contains info about the dependencies. On each cycle the processor examines the table and issues as many instructions as it can.

Instruction level parallelism (ILP): The number of instructions that can be executed simultaneously for a particular program and microarchitecture.

Register Renaming: Adds some nonarchitectural renaming registers to the processor (MIPS processor might add 20 renaming registers $\$r0 - \$r19$). The programmer cannot use these registers directly, because they are not part of the architecture, however the processor is free to use them to eliminate hazards. The processor keeps a table of which registers were renamed so that it can consistently rename registers insubsequent dependent instructions.

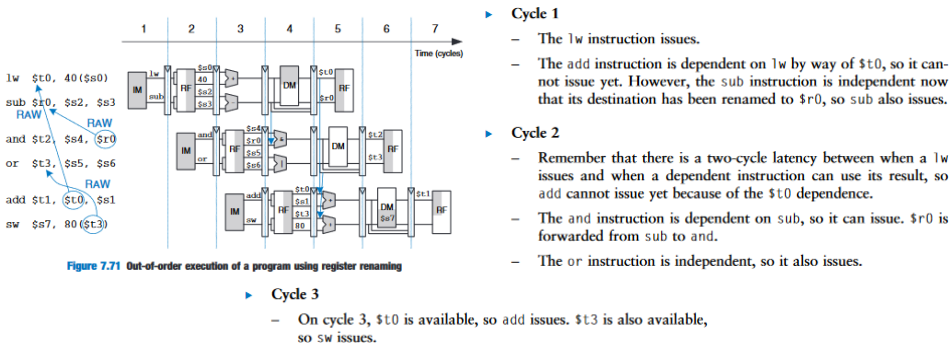
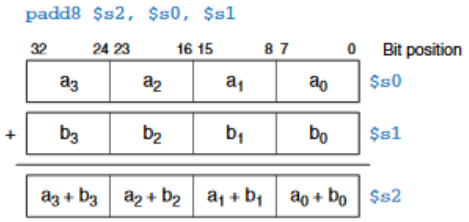


Figure 7.71 Out-of-order execution of a program using register renaming

It issues 6 instructions in three cycles for an IPC of 2

Single Instruction Multiple Data (SIMD): A single instruction acts on multiple pieces of data in parallel. e.g perform many short arithmetic operations at once ("packed arithmetic"). e.g pack 4 8-bit data elements into one 32-bit word. Packed add and subtract instructions operate on all four data elements within the word in parallel.



Performing packed arithmetic requires modifying teh ALU to eliminate carries between the smaller data elements i.e the carry out of $a_0 + b_0$ should not affect the result of $a_1 + b_1$

Multithreading: A technique that helps keep a processor with many execution units busy even if teh ILP of a program is low or the program is stalled waiting for memory. Terminology:

- **process:** A program running on a computer. Computers can run multiple processes simultaneously.
- **threads:** Each process consists of one or more threads which run simultaneously
- **Thread level parallelism (TLP):** The degree to which a process can be split into multiple threads that can run simultaneously
- **Context switching:** Threads take turns being executed on the processor under control of the OS. When one threads turn ends, the OS saves its architectural state, loads the architectural state of the next thread and starts executing that next thread. (Because of the speed of the switches, the user percieves all of the threads as running at the same time)
- **Multithreaded Processor:** contains more than one copy of its architectural state, so that more than one thread can be active at a time.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However it does improve the overall throughput of the processor.

Homogeneous Multiprocessors: A multiprocessor system consists of multiple processors and a method for communication between the processors. Homogeneous multiprocessing (symmetric multiprocessing (SMP)) is a form of multiprocessing in which two or more identical processors, share a single main memory. Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running a particular thread is challenging, the programmer must divide the existing thread into multiple threads to execute on each processor. Difficulty arises when the processors need to communicate with eachother. Homogeneous multiprocessors are good for situations, like large data centers, that have lots of thread level parallelism available.

cores: Multiple copies of the processor on the same chip

clustered multiprocessing: Each processor has its own local memory system. Clustering can also refer to a group of PC's connected together on the network running software to jointly solve a large problem.

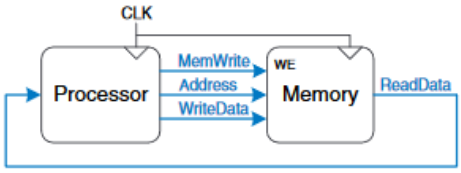
Heterogeneous Multiprocessors(asymmetric multiprocessors): Use seperate specialized microprocessors for separate tasks i.e incorporate different types of cores and/or specialized hardware in a single system. Each application uses tzhose resources that provide the best performance, or power-performance ratio. A heterogeneous system can incorporate cores with different microarchitectures that have different power, performance, and area tradeoffs. In such a system, the cores may all use the same ISA, which allows applications to run on any of the cores, or use different ISA's which can allow further customization of a core to given task. Heterogeneous systems are good for cases that have more varying workloads and limited parallelism. They add complexity in design and additional programming effort to combine the various resources.

Chapter 8

Memory and I/O Systems

8.1 Intro

Computer system performance depends on the memory system as well as the processor microarchitecture. Accessing memory is slow and can usually not be done in a single clock cycle. DRAM memories are currently 10 to 100 slower than processors. The processor communicates with the memory system over a **memory interface**:



The processor sends an address over the Address bus to the memory system. For a read, MemWrite is 0 and the memory returns the data on the ReadData bus. For a write, MemWrite is 1 and the processor sends data to memory on the WriteData bus.

Temporal locality: The processor is likely to access a piece of data again soon if it has accessed that data recently.

Spacial locality: When a processor accesses a piece of data, it is also likely to access data in nearby memory locations.

Memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data. Computer memories are primarily built from dynamic RAM and static RAM. In reality a memory is either slow, small or expensive. A computer system can approximate an ideal memory by combining a fast small cheap memory and a large slow cheap memory. The fast memory stores the most commonly used data and instructions, so on average the memory system appears fast. The large memory stores the remainder of the data and instructions, so the overall capacity is large.
The DRAM access time is one to two orders of magnitude longer than the processor cycle time. To counteract this, computers store the most commonly used instructions and data in a faster but smaller memory **cache**.

cache: Usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed, because SRAM is faster than DRAM and because on chip memory eliminates lengthy delays caused by traveling to and from a seperate chip. Caches can store both instructions and data.

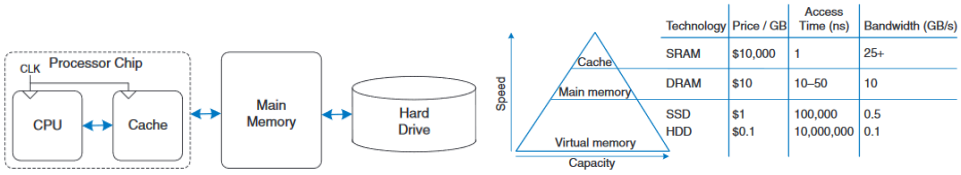
cache hit: If the processor requests data that is available in the cache, it is returned quickly.

cache miss: If the processor requests data that isnt available in the cache and the processor must retrieve data from main memory (DRAM)

Hard Drive: Computer systems use the hard drive to store data that does not fit in main memory. There are two types of Hard Drives:

- Hard Disk Drive (HDD)
- Solid State Drive (SSD): Built using flash memory. They overcome some of the mechanical failures of HDDs but costs 10 times as much.

The hard drive provides an illusion of more capacity than actually exists in the main memory hence it is also called "virtual memory". The main memory is also called "physical memory" and acts like a cache to the virtual memory.



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10–50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

8.2 Memory System Performance Analysis

Memory system performance metrics are miss rate or hit rate and average memory access time. Miss and hit rates are defined as:

$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Number of total memory accesses}} = \text{1- Hit Rate}$$

$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Number of total memory accesses}} = \text{1- Miss Rate}$$

Average memory access time (AMAT): The average time a processor must wait for memory per load or store instruction. AMAT is calculated as:

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM})$$

t's are the respective access times of the level of memory and MR is the respective miss rate.

8.3 Caches:

A cache holds commonly used memory data.

capacity: Number of data words a cache can hold.

When the processor atttempts to access data, it first checks the cache for the data. If the cache hitsm the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accomodate the new data, the cache must replace the old data. Caches use spatial and temporal locality to predict what data will be needed next.

What data is stored in Cache: The cache must guess what data will be needed based on the past pattern of memory accesses. The cache exploits temporal and spatial locality to achieve a low miss rate:

- Temporal: When the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache.
- Spatial: When the cache fetches one word from memory, it may also fetch several adjacent words (**cache block/cache line**).

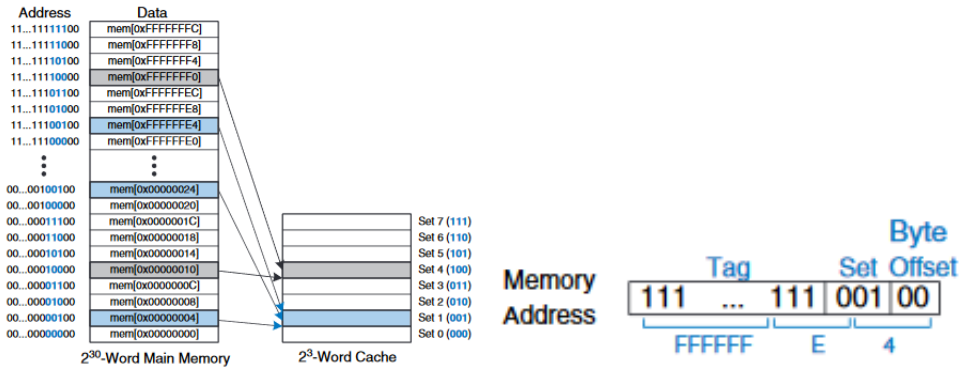
block size: The number of words in the cache block. A cache of capacity C contains:

$$B = \frac{C}{b}$$

How is Data Found A cache is organized into S sets, each of which holds one or more blocks of data.
mapping: The relationship between the address of data in main memory and the location of that data in the cache. Each memory address maps to exactly one set in the cache Caches are categorized based on the number of blocks in the set:

- direct mapped cache: Each set contains exactly one block i.e S=B sets.
- N-way set associative cache: Each set contains N blocks. i.e S $= \frac{B}{N}$ sets.
- Fully associative cache: Only has S=1 set.

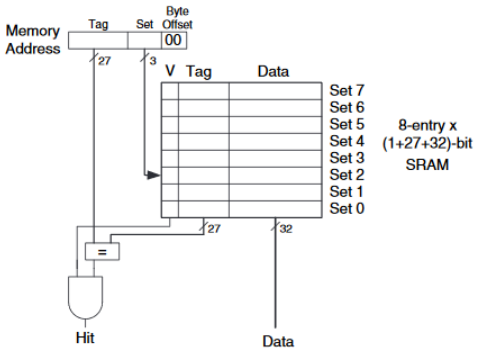
Direct Mapped Cache: The mapping of memory addresses onto cache works in a round-robin fashion i.e An address in block 0 of main memory maps to set 0 of the cache. Address in block 1 of main memory maps to set 1 of the cache. Address in block B of main memory maps to set 0 of the cache etc.



The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the **tag** and indicate which of the many possible addresses is held in that set.

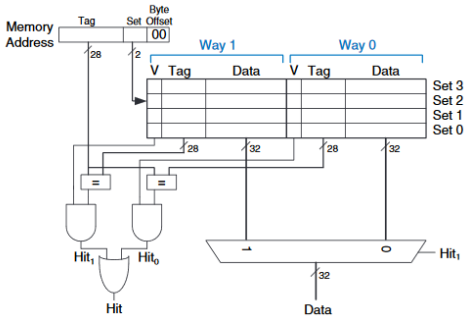
Valid bit: Indicates whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

The following image shows the hardware for the direct mapped cache.



The cache is constructed as an eight-entry SRAM. Each entry (set) contains one line consisting of 32 bits of data, 27 bits of tag and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next 3 bits (set bits) specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks the tag and valid bits. If the tage matches the most significant 27 bits of the address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise the cache misses and the memory system must fetch the data from main memory.
When two recently accessed addresses map to the same cache block a conflict occurs, and the most recently accessed address evicts the previous one from the block. Direct mapped caches have only on block in each set, hence two addresses that map to the same set always casue a conflict.

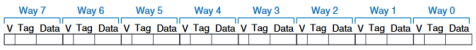
Multi-Way Set Associative Cache: An N-way set associative cache reduces conflicts by providing N blocks in each set where data mapping to that set might be found. Each memory address maps to a specific set, but it can map to any one of the N blocks in the set. N is also called the "degree of associativity" of the cache. Example: C=8-word, N=2 way set associative cache.



The cache has 4 sets rather than 8, hence only 2 set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two ways/degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways a mux selects data from that way. Set associative caches generally have lower miss rates than direct

mapped caches of the same capacity, because fewer conflicts occur. Set associative caches are usually slower and somewhat more expensive to build

Fully Associative Cache: Contains a single set with B ways, wher B is the number of blocks. A memory address can map to a block in any of these ways. Example: SRAM array of a fully associative cache with eight blocks.



Upon a data request, 8 tag comparisons must be made, because the data could be in any block. An 8:1 mux chooses the proper data if a hit occurs. Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size: To exploit spatial locality a cache uses larger blocks to hold several consecutive words. When a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched, hence subsequent accesses are morelikely to hit because of spatial locality. A large block size means that a fixed size cache will have fewer blocks which could lead to more conflicts and increase the miss rate. The time required to load the missing block into the cache is called the **miss penalty**. Example C=8-word direct mapped cache with a b=4 word block size.

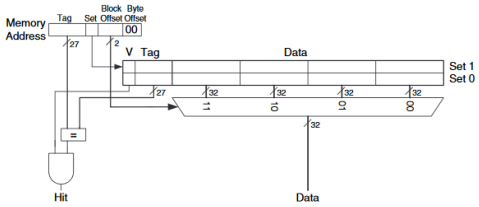
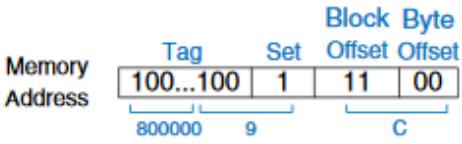


Figure 8.12 Direct mapped cache with two sets and a four-word block size



A mux is needed to select the word within the block, it is controlled by the 2 block offset bits of the address. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Putting it All Together: Caches are organized as 2-D arrays. The rows are called sets, and the columns are called ways.

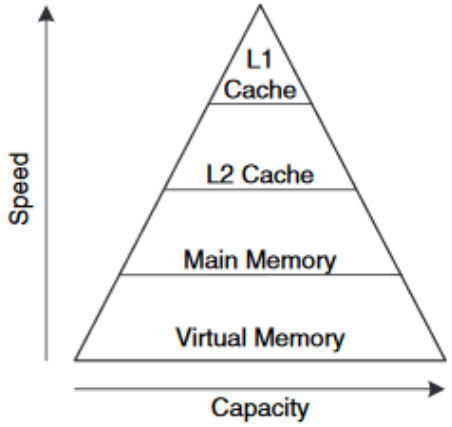
Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

- Increasing N reduces the miss rate caused by conflicts but increases hardware
- Increasing b reduces miss rate but reduces number of sets in a fixed cache and could lead to more conflict. Also increases miss penalty

What Data is Replaced: The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. This is called "least recently used" (LRU) replacement policy. In a two-way set associative cache, a "use" bit U indicates which way within a set was least recently used, when one of the ways gets used, U is adjusted to indicate the other way. For set associative caches with more than two ways, the ways are divided into two group and U indicates which group was least recently used. Upon replacement the new block replaces a random block within the least recently used group (pseudo-LRU).

Advanced Cache Design: Moder systems use multiple levels of caches to decrease memory access time.
Multiple-Level Caches: Large caches are beneficial because they are more likely to hold data of interest and have lower miss rates. However they tend to be slower than small ones. Modern systems use atleast 2 levels of cache:



L1-Cache is small enough to provide a one- or two-cycle access time.
Reducing Miss Rate: Cache misses can be reduced by changing:

- capacity
- block size
- associativity

Misses can be classified as:

- **Compulsory Miss:** The first request to a cache block must be read from memory regardless of the cache design
- **Capacity Miss:** When the cache is too small to hold all concurrently used data.
- **Conflict Miss:** When several addresses map to the smae set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss.

- Increasing cache capacity can reduce conflict and capcity misses, but does not affect compulsory misses
- Increasing block size could reduce compulsory misses (due to spatial locality) but might actually increase conflict misses (more addresses would map to the same set)
- Increasing associativity for small caches, decreases the number of conflict misses.

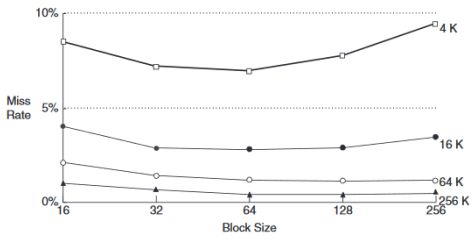


Figure 8.18 Miss rate versus block size and cache size on SPEC92 benchmark

Write Policy: Caches are classified as:

- **Write-Through:** The data written to a cache block is simultaneously written to main memory. A write-through cache requires no dirty bit but usually requires more main memory writes than write-back cache. (Hence modern caches are usually write-back)
- **Write-back:** A dirty bit is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache.

8.4 Virtual Memory

A hard disk (type of hard drive) contains one or more rigid disks or platters, each of which has a read/write head on the end of a long triangular arm. The head moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes seceral milliseconds to seek the correct location on the disk, which is millions of times slower than the processor. Adding a hard drive to the memory hierarchy gives the illusion of a very large memory while still providing the speed of faster memory for most accesses.
Example: 128MB of DRAM could provide 2GB of memory using the hard drive. The 2-GB memory is called **virtual memory** and the smaller 128-MB main memory is called **physical memory**
Location in virtual memory are specified with virtual addresses. The physical memory holds a subset of most recently accessed virtual memory i.e phyiscal memory acts as a cache for virtual memory. Virtual memory systems use different terms for the same caching principles:

Table 8.4 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

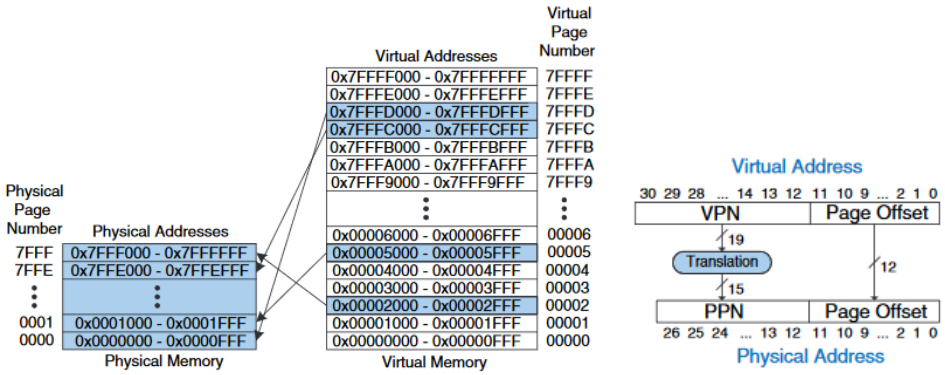
Virtual and Physical memory are divided into virtual and phyiscal pages typically 4KB in size. A virtual page may be located in physical memory or on the hard drive.

Address translation: The process of determining the physical address from the virtual address

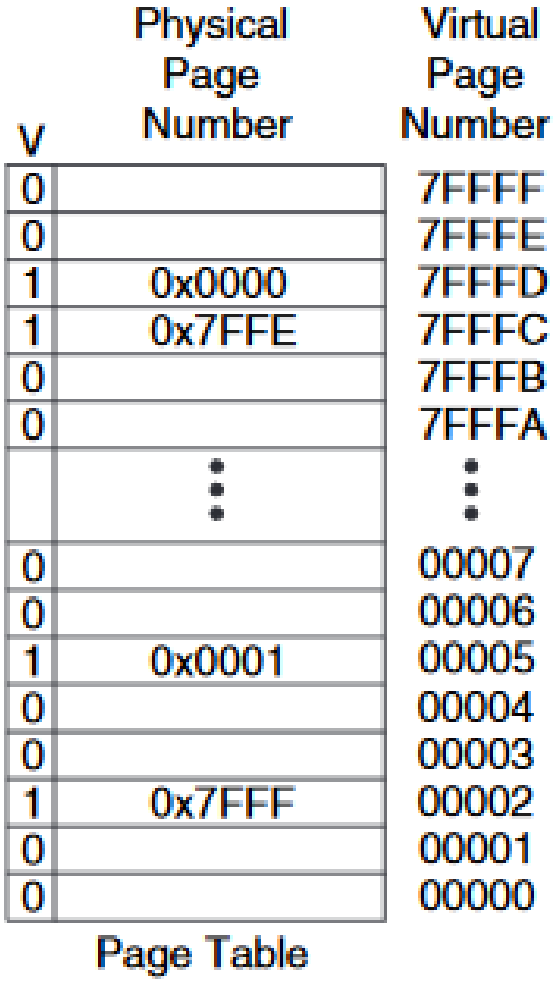
Page fault: If the processor attempts to access a virtual address that is not in physical memory, and the OS loads teh page from the hard drive into physical memory. To avoid page faults caused by conflicts, any virtual page can map to any physical page i.e physical memory behaves as a fully associative cache for virtual memory.
It would be too expensive for a virtual memory system to provide a comparator for each page. In practice the virtual memory uses a page table to perform address translation

Page Table: Contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data. To speed up address translation a translation lookaside buffer caches the most commonly used page table entries.

Address Translation: The most significant bits of the virtual or physical address specify the virtual or physical **page number**. The least significant bits specify the word within the page and are called the **page offset**
Example: 2GB virtual memory, 128 MB of physical memory divided into 4-KB pages.
With a 2-GB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used, the 32nd bit is always 0. Analogously with 128MB=2^{27}-byte physical memory, only the least significant 27 physical address bits are use, the upper 5 bits are always 0. The page size is 4KB=2^{12} bytes. The least significant 12 bits of the virtual and physical addresses are the same and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address. The right image below illustrates the translation of a virtual to a physical address. The upper 19 bits of the virtual address specify the virtual page number (VPN) nad are translated to a 15-bit physical page number (PPN)



The Page Table: The processor uses a page table to translate virtual addresses to physical adresses . It contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the bit is 1 the virtual page maps to the physical page specified in the entry. Otherwise the virtual page is found on the hard drive. Because the page table is so large, it is stored in physical memory.



Entry 6 is invalid (V=0) hence virtual page 6 is located on the hard drive.
The page table can be stored anywhere in physical memory at the discretion of the OS. The processor typically uses a dedicated register called the **page table register**, to store the base address of the page table in physical memory. To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally it reads or writes data at this physical address. Because the page table stored in physical memory each load or store involves two physical memory accesses.

The Translation Lookaside Buffer (TLB): The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. The processor can keep the last several page table entries in a small cache, the TLB. The processor looks in the TLB for the translation before having to access the page table in physical memory. A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If it hits, it returns the corresponding physical page number. Otherwise the processor must read the page table in physical memory. TLB is designed to be small enough that it can be accessed in less than one cycle.

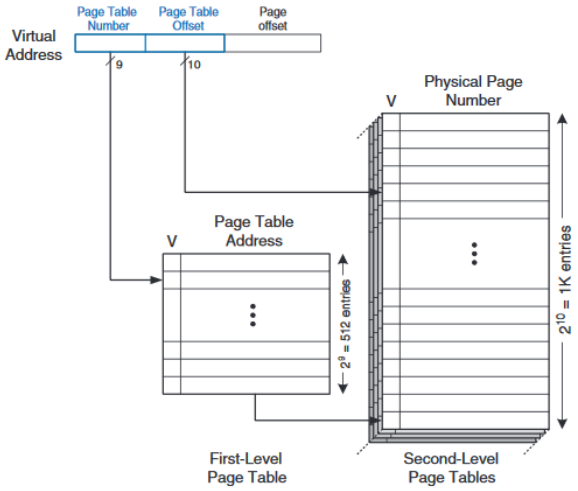
Memory Protection: Using virtual memory also provides protection between concurrently running programs. Memory protection is the principle that no program should be able to access another programs memory without permission. Virtual memory systems provide memory protection by giving each program its own virtual address space. A program can access only those physical pages that are mapped in its page table. This way, a program cannot accidentally or maliciously access another programs physical pages, because they are not mapped in its page table. The OS adds control bits to each page table entry to determine which programs, can write to the shared physical pages.

Replacement Policies: Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from the physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called **paging**. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that

page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

Multilevel Page Tables: To conserve physical memory, page tables can be broken up into multiple levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. In a two-level page table, the virtual page number is split into two parts:

- **page table number:** indexes the first-level page table
- **page table offset:** indexes the second-level page table



The two-level page table requires a fraction of the physical memory needed to store the entire page table but it requires an additional memory access for translation when the TLB misses.

Chapter 9

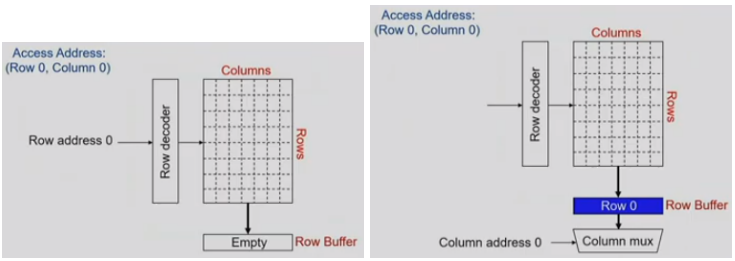
Summary of Lecture Slides

9.1 Many Cores on Chip

Ideally: N times system performance with N times the cores. (linear scaling)
Reality: Unexpected Slowdowns in Multi-Core system. When running applications on a multi-core system some applications have a much larger slowdown when ran by themselves

Why the Disparity in Slowdowns?
The example is using Matlab and gcc running on 2 cores of a Multi-Core Chip. They have a Shared DRAM Memory system. In this case matlab is an aggressive program i.e it generates alot of requests. The Memory Controller prioritizes the requests from matlab, hence there is alot more progress in the matlab being done than in gcc.

DRAM Bank Operation: A DRAM Bank is built from a matrix of DRAM cells, connected to a **Row Buffer** which is essentially a cache for the row being accessed. To access data from a row it must first be loaded into the row buffer.



The Row decoder is responsible for activating the row which will be accessed. upon activation the row is loaded into the row buffer. The memory controller can now provide the column address. The specified data is then forwarded to the processor. For the following requests there are two possibilities:

- **Hit:** If the the row is already loaded in the row buffer the controller skips the row activation part and just reads the column address. ⇒ Operation takes much less time
- **Conflict:** When the row loaded in the row buffer doesnt match the row specified in the access request. Before activiting the requested row it closes ("precharges") the current row in the row buffer (which costs additional time)

A row-buffer conflict consumers more energy than a row-buffer hit. A conflict requires a precharge, an activate and a read/write, whereas a hit only requires a read/write Commonly used scheduling policy:

- **Row-hit-first:** Requests that are hit in the buffer are prioritized
- **Oldest-first:** Second set of rows that are prioritized are the ones that arrived first

⇒ This leads to scheduling unfairness.

DRAM scheduling policies are unfair to some applications:

- **Row-hit first:** unfairly prioritizes apps with high row buffer locality (threads that keep accessing the same row)
- **Oldest first:** unfairly prioritizes memory intensive applications

9.2 Lecture 15: Out of Order Execution

The problem with in order execution is that the pipeline might get stalled, because it is waiting for the results of the previous instruction because the current one is dependent on its results. During this time independent instructions could proceed, hence we have a loss in performance.
We can solve this by moving the dependant instructions out of the way of independent ones such that the independent ones can execute. The dependent instructions are moved to "reservation stations" where the source values are monitored.

When all source values of an instruction are available we "fire" (i.e dispatch) the instruction
⇒ Instructions dispatched in dataflow order (not control-flow) The main benefit is latency tolerance i.e independent instructions can execute and complete in the presence of a long latency operation.

Precise Exceptions: If an exception appears all operations before the exception are completed i.e that data of previous operations are saved into registers so that there is no data loss.

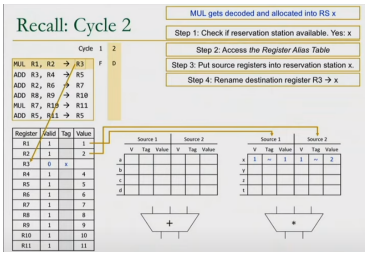
Enabling OoO execution:

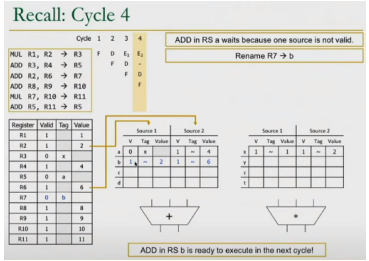
1. link the consumer (i.e an instruction which needs the value) of a value to the producer (instruction that produces the value).
   ⇒ Register renaming: Associate a tag with each data value. We use the tag to identify the instruction which will produce that value.
2. Buffer instructions until they are ready to execute (take them out of the way until they are ready to be used) ⇒ insert instruction into reservation stations after renaming
3. Keep track of readiness of source values
   ⇒ Broadcast the tag when the value is produced. Instructions compare their source tags to the broadcast tag, if there is a match, the source value becomes ready
4. when all source values of an instruction are ready, dispatch the instruction to its functional unit ⇒ Intstruction wakes up if all sources are ready, if multiple are awake we need to select one per FU

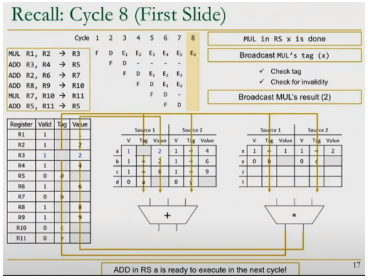
Register Renaming Output and anti dependencies are not true dependencies because the same register refers to values that have nothing to do with each other. They exist because there are not enough register ID's in the ISA. The register ID is renamed to the reservation station entry that will hold the register's value. This eliminates anit- and output- dependencies.

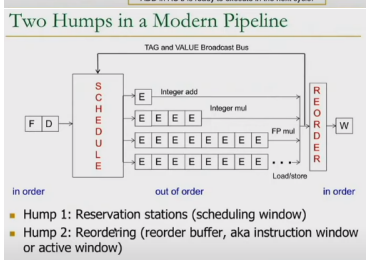
Tomasulo's Algorithm:

- If a reservation station is available before renaming, then we insert the instruction+ renamed operands into the reservation station. If there is no station available then we stall
- While in the reservation station each instruction watches the common data bus (CDB) for tag of its sources, when the tag is seen, it grabs the value for the source and keeps it in the reservation station. When both operands are available the instruction is ready to be dispatched
- Dispatch instruction to the FU when instruction is ready
- After the instruction finishes in the FU, we put the tagged value onto the CDB (tag broadcast). The register file is connected to the CDB and contains a tag indicating the latest writer to the register. If the tag in the register file matches the broadcast tag, we write the broadcast value into the register (and set valid bit). We then reclaim the rename tag (so no valid copy of tag in the system)









Register rename table(register alias table): A register can be valid/invalid. If its valid you get the value out of the register alias table. If its invalid it gets produced by the instruction which has a tag.

Dataflow Graph: A dataflow graph has nodes (operations) and arcs (values needed by the operations)

9.3 Lecture 16b + 17: Branch Prediction

Branch Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

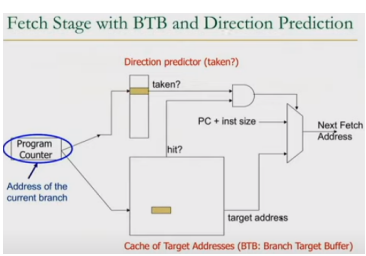
How to Handle control Dependences
If control-flow instructions:

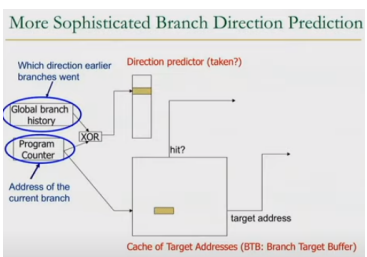
- Stall the pipeline until next fetch address is known. Not advisable because CPI increases
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot). Change the semantics of the ISA i.e branch takes effect after the next N instructions have executed. E.g if pipeline is 4 stages you can always put independent instructions in the delay slot and they will always be executed.
- Do something else (fine-grained multithreading). Multiple threads can be fetched from i.e each cycle we fetch from a different thread. If there are more threads than pipeline stages then no prediction is necessary because pipeline is always full.
- Eliminate control-flow instructions (predicted execution)
- Fetch from both possible paths (if you know the addresses of both possible paths)(multipath execution). This is very costly

The Branch Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor. N is called the minimum branch resolution latency.
If we are fetching W instructions per cycle i.e if pipeline is W wide then a branch misprediction leads to $N \cdot W$ wasted instruction slots.
Methods to predict next PC:

- Always predict the next sequential instruction is the next instruction to be executed (accuracy of 30%)
- Maximize the chances that the next sequential instruction is the next instruction to be executed i.e we lay out the control flow of the graph such that the "likely next instruction" is on the not-taken path of a branch. This means that the compiler will order the code in such a way that the block of code which follows the branch will be put next to the code before the branch in order to increase the sequential section of code i.e most frequent transitions are consecutive in memory.
  How do compilers know how often a branch is taken?
  ⇒ **profiling:** The program is run multiple times and we observe how many times the branch was taken. The danger is that the profiling could be false and result in bad performance, hence profiling is subject to the representative of the input set.
- Get rid of control flow instructions (or minimize their occurence) by:
  - Get rid of unnecessary control flow instructions i.e combine predicate
  - Convert control dependences into data dependences i.e predicated execution
- Predict the next fetch address (to be used in the next cycle). This requires 3 things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch (can be done with a BTB)
  - (conditional) branch direction
  - Branch target address (if taken) (can be done with a BTB)

  We know the target address remains the same for a conditional direct branch across dynamic instances. We store the target address from the previous instance and access it with the PC. This is called **Branch Target Buffer (BTB)** or branch Target Address Cache





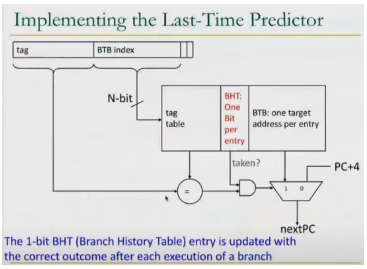
Static Branch Prediction:

- **Always not-taken:** Simple to implement, no need for BTB and no direction prediction (PC = PC + 4). This method has low accuracy $\sim 30 - 40\%$(for conditional branches)
- **Always taken:** Assume always take a branch. Better accuracy $\sim 60 - 70\%$ because backward branches (target address lower than branch PC) are usually taken
- **Backward taken, forward not taken:** Predict backward branches as taken, others as not-taken
- **Profile Based:** Compiler determines likely direction for each branch using a profile run. Encodes that direction as a hint bit in the branch instruction format. Compared to the earlier methods this method actually distinguishes between different branches which leads to higher accuracy but also requires an additional bit.
- **Program-based** Use heuristics based on program analysis to determine statically predicted direction
- **Programmer-based** Programmer provides the statically predicted direction. Via pragmas in the programming language that qualify a branch as likely-taken versus likely not taken. This method does not require profiling or program analysis and has insight of the programmer which can be communicated. But this method requires programming language,compiler and ISA support.

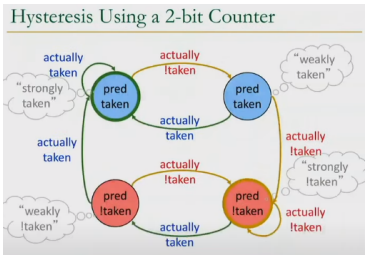
The disadvantage to these techniques is that they cannot adapt to dynamic changes in branch behavior **Pragmas** Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

Dynamic Branch Prediction:

- **Last time predictor:** Single bit per branch stored in BTB, which indicates which direction the branch went last time it was executed. This always mispredicts the last iteration and the first iteration of a loop branch, hence it is very effective for large loop sizes



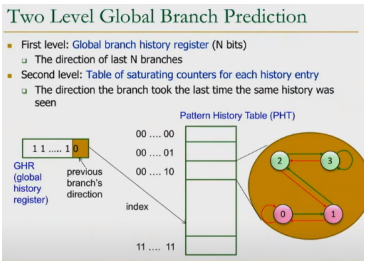
Problem: Last time predictor changes its prediction too quickly.
Solution: Add hysteresis to the predictor so that prediction does not change on a single different outcome i.e use two bits to track the history of predictions for a branch instead of a single bit.

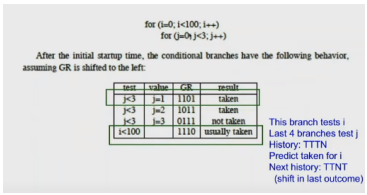


This leads to two realizations:

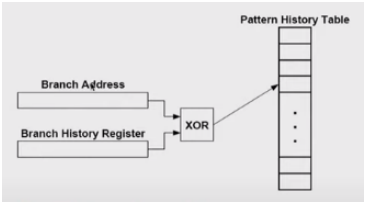
1. A branch's outcome can be correlated with other branches outcomes i.e global branch correlation
2. A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)

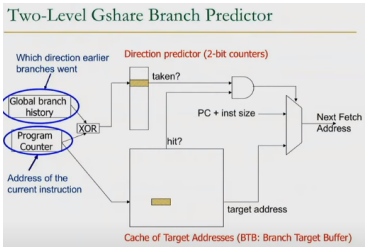
Global Branch Correlation Recently executed branch outcomes in the execution path are correlated with the outcome of the next branch. The idea is to associate branch outcomes with "global T/NT history" of all branches and make a prediction based on the outcome of the branch the last time the same global branch history was encountered. This is implemented by keeping tack of the global T/NT history of all branches in a register (Global History Register (GHR)). The GHR is used to index into a table that recorded the outcome that was seen for each GHR value in the recent past, the Pattern History Table(table of 2-bit counters).



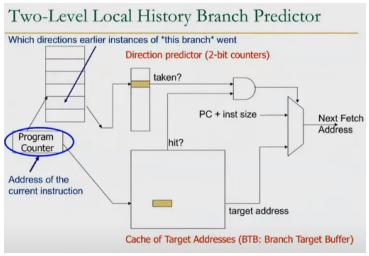


We can improve global predictor accuracy by adding more context information to the global predictor to take into account which branch is being predicted.
⇒ Gshare predictor: GHR hashed with the Branch PC. We do not only rely on the branch history register to make the prediction but also on the branch address.



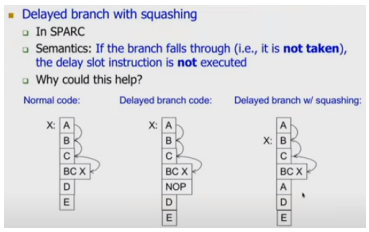


To predict a loop branch perfectly we want to identify the last iteration of the loop i.e when it switches from taken to not taken. We achieve this by having a seperate Pattern History Table (PHT) entry for each local history. This works for short loops where the bit length of the entry is larger equal to the pattern length.
⇒ we have a per-branch history register to associate the predicted outcome of a branch with T/NT history of the same branch i.e make a prediction based on the outcome of the branch the last time the same local branch history was encountered. This is called **local history/branch predictor**



Delayed Branching: Delay the execution of a branch. N instructions that come after the branch are always executed regardless of branch direction. Instructions which can fill the delay slots:

- Branch must be independent of delay slot instructions



Local Correlation: For a specific branch if given the previous value i.e T or N can i decide whether or not to take the branch

9.4 Lecture 18a: VLIW (Very Long Instruction Word)

superscalar Hardware fetches multiple instructions and checks dependencies between them

VLIW Software (compiler) packs independent instructions in a larger "instruction bundle" to be fetched and executed concurrently. The hardware fetches and executes the instructions in the bundle concurrently hence no need for hardware dependency checking between concurrently fetched instructions in the VLIW model. Packed instructions can be logically unrelated from eachother.

The idea is that the compiler finds independent instructions and statically schedules (i.e packs/bundles) them into a single VLIW instruction. Characteristics:

- Multiple functional units
- All instructions in a bundle are executed in lock step (i.e one is completed before the next on starts)
- Instructions in a bundle are statically aligned to be directly fed into the functional units (no additional hardware necessary)

If the compiler cant find enough independent instructions to create a bundle it inserts a NOP (no operation). If any operation in a VLIW instruction stalls, all instructions stall.

VLIW Tradeoffs

- Advantages:
  - No need for dynamic scheduling hardware → simple hardware
  - No need for dependency checking within a VLIW instruction → simple hardware for mutliple instruction issue and no renaming
  - No need for instruction alignment after fetch to different functional units → simple hardware
- Disadvantages:
  - Compiler needs to find N independent operations per cycle
  - Recompilation required when execution width (N)m instruction latencies or functional units change
  - Lockstep execution causes independent operations to stall i.e no instruction can progress until the longest latency instruction completes.

9.5 Lecture 18b: Systolic Arrays and Beyond

Systolic Arrays: The goal is to design an accelerator (i.e excels at a certain task) which has a simple and regular design, highly concurrent and a balanced computation and I/O bandwidth. The idea is to replace a single processing element (PE) which could be a FU or a processor with a regular array of PE's and carefully orchestrate the flow of data between the PE's such that they collectively transform a piece of input data before outputting it to memory (we can perform lots of different operations which need the same data and only need to fetch it once)

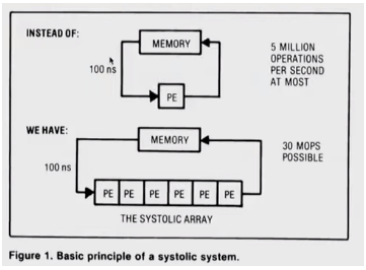


Figure 1. Basic principle of a systolic system.

The Systolic array can be multi-dimensional and the PE connections can also be multidirectional with different speeds. The PE's can have local memory and execute kernels

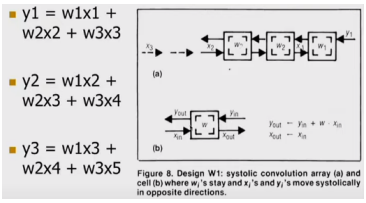


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

Systolic arrays can be chained together to form powerful systems.

Pros and Cons: Pros:

- Principles: Efficiently makes use of limited memory bandwidth, balances computation to I/O bandwidth availability
- specialized (computation needs to fit PE organization/functions).
- makes mutliple uses of each data item i.e reduced need for fetching/refetching i.e better use of memory bandwidth

The Con is that it is specialized because its not generally applicable, the computation needs to fit the PE functions/organization.

9.6 Lecture 19: SIMP Processing:

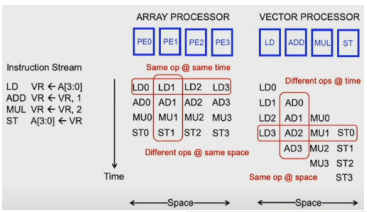
types of computers:

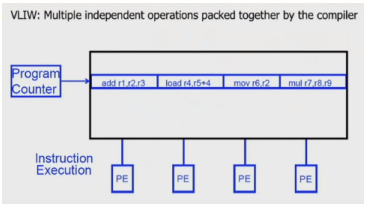
- **SISD** Single instruction operates on single data element
- **SIMD** Single instruction operates on multiple data elements (array processor, vector processor)
- **MISD** Multiple instructions operate on single data element (systolic arrays)
- **MIMD** Multiple instructions operate on multiple data elements (multi core system)

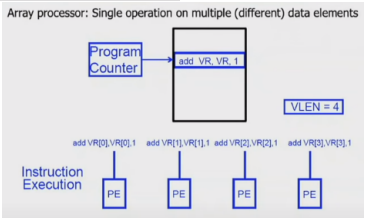
Concurrency arises from performing the same operation on different pieces of data

SIMD: SIMD exploits operation level parallelism on different data. We have two types of processors:

- **Array Processor:** Instruction operates on multiple data elements at the same time using different spaces
- **Vector Processor:** Instruction operates on multiple data elements in consecutive time steps using the same space.







Vector: A vector is a one-dimensional array of numbers. A vectorizable loop is a loop for which each iteration is independent. A vector processor is one whose instructions operate on vectors rather than scaler (single data) values. Requirements:

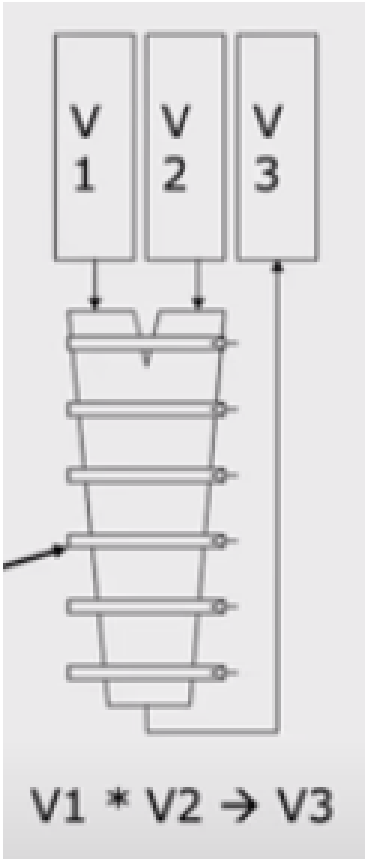
- Need to load/store vectors with vector registers
- Need to operate on vectors of different lengths → vector length register (VLEN)
- Elements of a vector might be stored apart from eachother in memory → vector stride register (VSTR). the stride is the distance in memory between two elements of a vector

A vector instruction performs an operation on each element in consecutive cycles. The vector functional units are pipelined and each pipeline stage operates on a different data element. Vector instructions allow deeper pipelines because:

- No Intra-vector dependencies → no hardware interlocking needed within a vector
- No control flow within a vector because its one single instruction i.e no branches
- known stride allows easy address calculation for all vector elements (allows prefetching of vectors into registers/cache/memory)

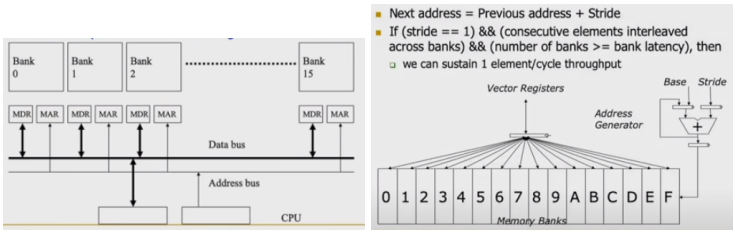
Vector Registers: Each vector data register holds N M-bit values. The vector control registers are: VLEN, VSTR and VMASK. VLEN can be N and indicates the max number of elements stored in a vector register. Vector Mask Register (VMASK) indicates which elements of the vector to operate on (e.g for conditional code).

Vector Functional Units: Use a deep pipeline to execute element operations. They have a fast clock cycle. Because the elements in the vector are independent control is simple.

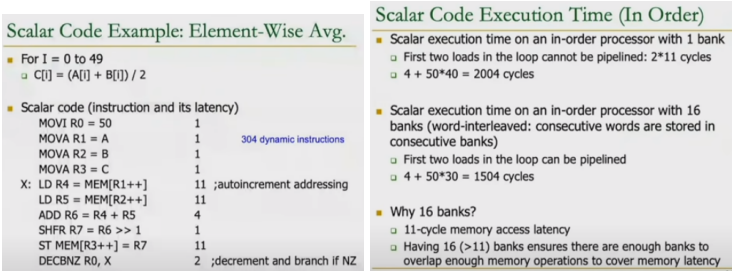


Loading/storing Vectors from/to Memory: Requires the loading/storing of multiple elements. The elements are separated from each other by a constant distance (stride). Elements can be loaded in consecutive cycles if we can start the load of one element per cycle. How is this done, since memory takes more than 1 cycle to access? The solution is we **bank** the memory

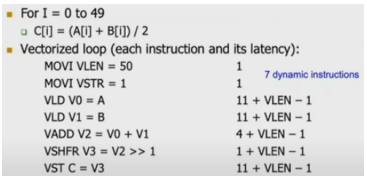
Memory Banking: Memory is divided into banks that can be accessed independently. Banks share address and data buses. We can start and complete one bank access per cycle and sustain N parallel accesses if all N go to different banks.



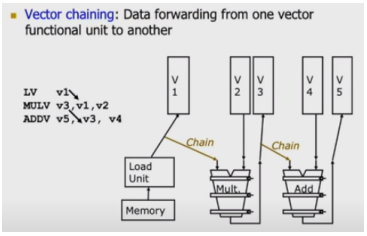
Bank Conflict: When the stride is picked such that there are multiple data in the same bank, hence we lose parallelism



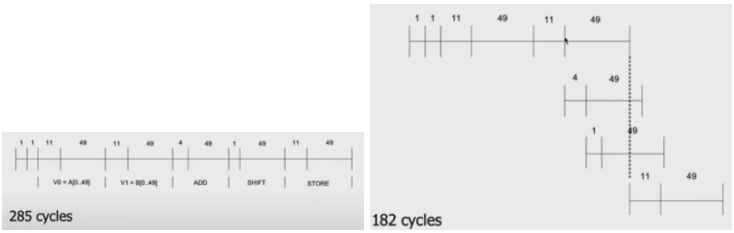
Vectorizable Loops: A loop is vectorizable if each iteration is independent of any other.



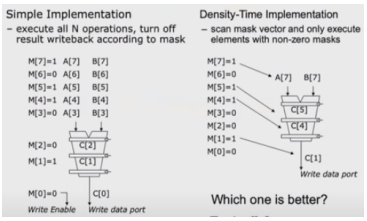
vector chaining: vector data forwarding i.e output of a vector functional unit is used as the direct input of another



We compare the number of cycles between the vector codes with and without vector chaining:



Because we only have one MAR and one MDR per data bank we cannot have multiple memory accesses in parallel.



Masked Vector Instructions: The Density-Time implementation is more efficient, as we only operate on the necessary data elements. On the otherhand because we need to scan the array we increase the latency of each operation.

9.7 Lecture 20: Graphics Processing Units:

Fine-Grained Multithreading: Hardware has multiple thread contexts. Each context has a PC and registers. Each cycle, the fetch engine fetches from a different thread. This means no two instructions from a thread are in the pipeline concurrently

GPU's are SIMD Engines underneath. The instruction pipeline operates like a SIMD pipeline (e.g array processor) however the programming is done using threads not SIMD instructions.

- Each thread executes the same code but operates on a different piece of data
- Each thread has its own context i.e can be treated restarted/executed independently

A set of threads executing the same instruction are dynamically grouped into a warp (wave front) by the hardware. A warp is essentially a SIMD operation formed by hardware

Programming Model: Refers to how the programmer expresses the code. e.g Sequential, data parellel, data flow etc.

Execution Model: Refers to how the hardware executes the code underneath. e.g OoO execution, vector processor, array processor, dataflow processor, Multiprocessor, etc
⇒ The execution model can be very different from the porgramming model

SIMD vs. SIMT execution model:

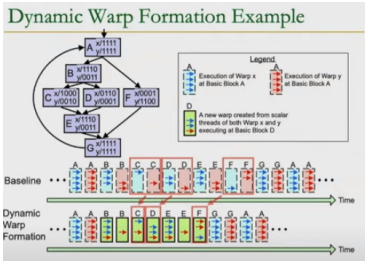
- **SIMD:** A single sequential instruction stream of SIMD instr. → each instruction specifies multiple data inputs
- **SIMT:** Multiple instruction streams of scalar instructions → threads grouped dynamically into warps

Two major SIMT advantages:

- Can treat each thread separately i.e can execute each thread independently
- Can group threads into warps flexibly

Branch Divergence: Threads inside warps branch to different execution paths.

Dynamic Warp formation: Partial warps that dont have threads running in the same slot can be merged:



9.8 NVIDIA TESLA:

Vertex operators: Operate on the vertices of primitives i.e points,lines, triangles. Ops include transforming coordinates into screen space and feeding to the setup unit, setting up lighting and texture parameters to be used by pixel fragment processors. Designed for low-latency, high precision math operations

Pixel-fragment processor: operate on rasterizer output, which fills the interior of primitives. Designed for high-latency, low precision texture filtering

GPU's typically process more pixels than vertices,hence pixel fragment processors outnumber vertex processors ($\sim 3:1$). However typical workloads are unbalanced.
The design objective for Tesla was to execute vertex and pixel-fragment shader programs on the same unified processor architecture. Unification enables dynamic load balancing of vertex and pixel-processing workloads.

Tesla Architecture: Based on a scalable processor array. E.g GeForce 8800 GPU:

- 128 Streaming-processor (SP) cores
- SP cores organised into 16 streaming multiprocessors (SM's)
- SM's contained in 8 independent processing units (texture/processor clusters (TPC's))

The Scalable streaming processor array (SPA) performs all the GPU's programmable calculations. The scalable memory system consists of external DRAM control and fixed-function raster processors (ROP's) which perform color and depth fram buffer operations directly on memory. The interconnection network carries computed pixel-fragment colors and depth values from the SPA to the ROP's and routes memory read requests from the SPA to DRAM and read data from DRAM through a level-2 cache back to the SPA. The input assembler collects vertex work as directed by the input command stream.

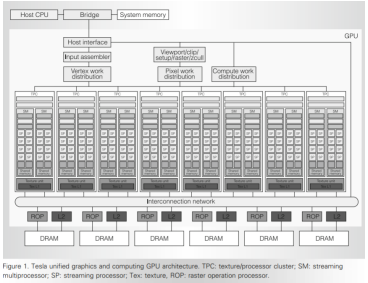


Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

Command Processing: The GPU **host interface** unit communicates with the host CPU, responds to commands from the CPU, fetches data from system memory,...
Input Assembler: collects geometric primitives (points, lines, triangles) and fetches associated vertex input attribute data.
Work Distribution units: forward the input assembler's output stream to the array of processors, which execute vertex, geometry, and pixel shader programs, as well as computing programs.
Vertex and compute work distribution units deliver work to processors in a round robin scheme.

Streaming processor Array: SPA executes graphics shader thread programs and GPU computing programs and provides thread control and management. The number of TPC's determines a GPU's processing performance.

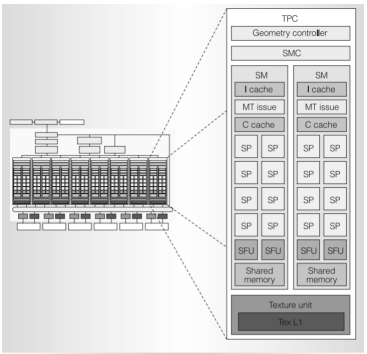
Vertex Shader: processes one vertex's attributes independently of other vertices. (ops include position space transforms and color and texture coordinate generation.

Geometry shader: Deals with a whole primitive and its vertices. (ops include edge extrusion, and cube map texture generation)

Texture/processor cluster: Each TPC contains:

- **Geometry controller:** maps the logical graphics vertex pipeline into recirculation on the physical SM's by directing all primitive and vertex attribute and topology flow in the TPC. It manages dedicated on-chip I/O vertex attribute storage and forwards contents as needed.
- SM controller
- **2 streaming multiprocessors:** The SM is a unified graphics and computing multiprocessor that executes vertex,geometry, and pixel-fragment shader programs and parallel computing programs. The SM consists of:
  - 8 streaming processor cores: Each of these cores contains a scalar multiply-add (MAD) unit
  - 2 Special function units: For transcendental functions and attribute.
  - 1 multithreaded instruction fetch and issue unit (MT Issue)
  - Instruction cache
  - Read-only constant cache
  - 16-Kbyte read/write shared memory: holds graphics input buffers or shared data for parallel computing

  To pipeline graphics workloads through the SM, vertex, geometry and pixel threads have independent input and output buffers hence workloads can arrive and depart independently of thread execution.
- **Texture unit** used by the SM as a third execution unit,



igure 2. Texture/processor cluster (TPC).

SM Multithreading: A **graphics vertex or pixel shader** is a program for a single thread that describes how to process a vertex or a pixel.
A **CUDA kernel** is a C program for a single thread that describes how one thread computes a result.
In order to dynamically balance shifting vertex and pixel shader thread workloads, the unified SM concurrently executes different thread programs and different types of shader programs. The SM is hardware multithreaded and can execute 768 concurrent threads in hardware with zero scheduling overhead. Each SM thread has its own thread execution state and can execute an independent code path. Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction.

Single-instruction, multiple thread: In order to manage and execute hundreds of threads running several different programs efficiently, the Tesla SM uses Single Instruction Multiple Thread (SIMT) architecture. The SM's SIMT multithreaded instruction unit creates, manages, schedules and executes threads in groups of 32 parallel threads called **warps**. Each SM manages a pool of 24 warps. Individual threads composing a SIMT warp are of the same type and start together at the same program address, but are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute adn issues the next instruction ot that warp's active threads. A SIMT instruction is broadcast synchronously to a warps active parallel threads (individual threads can be inactive due to independent branching or predication). The SM maps the warp threads to the SP cores and each thread executes independently with its own instruction address and register state. Full efficiency and performance is reached when all 32 threads of a warp take the same execution path. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete the threads reconverge to the original execution path. Branch divergence only occurs within a warp, different warps execute independently regardless of whether they are commmon or disjoint code paths. The difference between SIMT and SIMD:

- SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes.

- SIMD instruction controls a vector of multiple data lanes together and exposes the vector width to the software, SIMT instruction controls the execution and branching behavior of one thread.
- SIMT enables programmers to write thread level parallel code for independent threads as well as data parallel code for coordinated threads.

SIMT warp scheduling The SM schedules and executes multiple warp types concurrently. At each cycle it selects one of the 24 warps to execute a SIMT warp instruction. An issued warp instruction executes as two sets of 16 threads over four processor cycles. The SP cores and SFU units execute instructions between them on alternate cycles, the scheduler can keep both fully occupied.

SM instructions: The Tesla SM executes scalar instructions. Scalar instructions are simpler and compiler firendly. Texture instructions remain vector based, takng a source coordinate vector and returning a fitered color vector.

Memory access Instructions: For computing, load/store instructions access three read/write memory spaces:

- **local:** memory for per-thread, private, temporary data
- **shared:** memory for low-latency access to data shared by cooperating threads in the same SM
- **global:** memory for data shared by all threads of a computing application

Computing programs use the fast barrier synchronization instruction to synchronize threads within the SM that communicate with each other via shared and global memory.

Streaming Processor (SP): Is the primary thread processor in the SM. It performs the fundamental floating-point operation and also a wide variety of integer, comparison and conversion operations. The unit is fully pipelined, and latency is optimized to balance delay and area.

Special-function unit (SFU): Supports computation of both transcendental functions and planar attribute interpolation. The SFU unit generates one 32-bit floating point result per cycle.

SM controller: The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simultaneously:

- Vertex
- Geometry
- Pixel

SMC packs each of these input types into the warp width, initiating shader processing and unpacks the results. Each input type has independent I/O paths, but the SMC is responsible for load balancing among them.

Texture Unit: processes one group of 4 threads per cycle. Texture instruction sources are texture coordinates and the outputs are filtered samples, typically a four-component (RGBA) color. Texture is a seperate unit external to the SM connected via the SMC. The issuing SM thread can continue execution until a data dependency stall. The texture unit is deeply pipelined.

9.9 Lecture 21a: Memory organization and Memory technology

Programmers View: Memory is a black box, we store and load data. The programmer sees **virtual memory** i.e the programmer assumes the memory is infinite. Reality is the physical memory size is much smaller than what the programmer assumes.
⇒ The system (software+ hardware) maps the virtual memory addresses to physical memory. The advantage being that the programmer does not need to know the physical size of memory nor manage it. The downside is that software and architecture becomes more complex

Physical Memory System: The physical memory has a backing store, the disk. When the programmer needs data which is not in physical memory we load it from the disk.

DRAM is on a seperate chip from the processor because it contains a capacitor. When accessing data from DRAM we first send n bits to decode the row and then send m bits to decode the column. By doing so we reduce the number of pins needed on the DRAM chip and maintain a decent speed. SRAM on the other hand is on the same chip as the processor and does not rely on pins hence we can send m+n bits at once.
DRAM vs. SRAM:

- DRAM
  - slower access (capacitor)
  - higher density (1T 1C cell)
  - lower cost
  - requires refresh
  - manufacturing requires putting capacitor and logic together
- SRAM

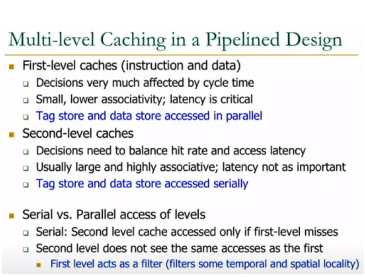
- Faster access (no capacitor)
- Lower density (6T cell)
- Higher cost
- No need for refresh
- Manufacturing compatible with logic process

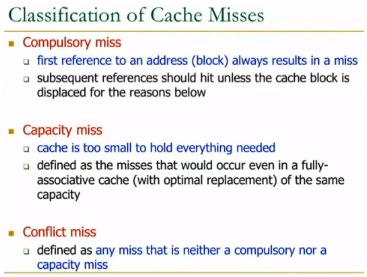
9.10 Lecture 22: More Caches:

Instruction vs. Data Caches: The pros and cons of unified:

- + Dynamic sharing of cache space means no overprovisioning that might happen with static partitioning (seperate I and D caches)
- - Instructions and data can thrash each other (i.e no guaranteed space for either)
- - I and D are accessed in different places in the pipeline hence where to place unified cache for fast access

In the fetch stage the program counter needs to access the instructions so the instruction cache is placed there. Data is accessed after you generate the address of a load/store which is another part of the pipeline. L1 caches are usually integrated into the pipeline.





Caches in Multi-Core Systems: We different between two types of caches:

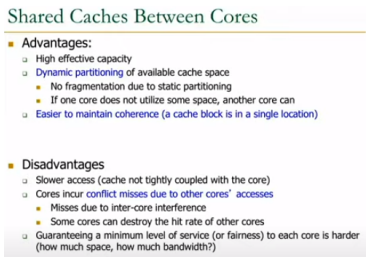
- **Private cache:** Cache belongs to one core
- **Shared cache:** Cache is shared by multiple cores

Advantages of resource sharing:

- Resource sharing improves utilization/efficiency i.e when a resource is left idle by one thread another thread can use it; no need to replicate shared data
- Reduces communication latency: e.g data shared between multiple threads can be kept in the same cache in a multithreaded processor

Disadvantages:

- Resource sharing results in contention for resources i.e when the resource is not idle, another thread cannot use it
- Sometimes reduces each or some threads performance
- Eliminates performance isolation i.e inconsistent performance across runs as thread performance depends on co-executing htreads



Cache Coherence: If multiple processors cach the same block how do they ensure they all see a consistent state?

9.11 Lecture 23a: Multiprocessor Caches

How to improve Cache performance:

- Reducing miss rate (reducing miss rate can reduce performance if more costly to refetch blocks are evicted)
- Reducing miss latency or miss cost

- Reducing hit latency or hit cost

There is a tradoff between the 3 parameters e.g decreasing cache size to improve speed will increase miss rate.

software approach for higher hit rate:

- Restructure data access paterns:
- Restructure data layout
- Loop Interchange: switch loop iteration pattern depending on row or column major
- Blocking: Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache i.e divide working set so each piece fits in the cache

Caches in Multi-Core Systems:

Advantages of shared caches:

- High effective capacity
- Dynamic partitioning of available cache space i.e no fragmentation due to static partitioning. If one core does not utilize some space another core can
- Easier to maintain coherence i.e if cache is updated all cores see change

Disadvantages:

- slower access
- Cores incure conflict misses due to other cores accesses
- Guaranteeing a minimum level of service (or fairness) to each core is harder

cache coherance: If multiple processors cache the same block, how do they ensure they all see a consistent state? e.g if two caches load teh same data and one updates it. now the other loads the data again, which value should it see. It should see the updated one. One appraoch would be to update all processors with the same cache block when one processor updates a value. ANother solution would to invalidate other caches.

Hardware Cache Coherence: A processer/cache broadcasts its write/update to a memory location to all other processors. Another cache that has the location either updates or invalidates its local copy. The two major approaches:

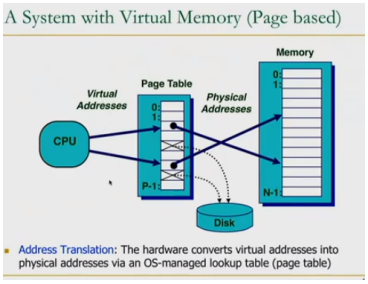
- Snoopy bus (all operations are broadcast on a shared bus)
- Directory based (a mediator gives permission to each request)

9.12 Lecture 23b + 24: Virtual Memory

The programmer sees virtual memory i.e they can assume the memory is infiniite. Reality is the physical memory size is much smaller than what programmer assumes. The system maps virtual memory addresses to physical memory. This way the programmer does not need to know the physical size of memory nor need to manage it. This requires more complex system software and architecture.

Benefits of Automatic Management of Memory

- Programmer does not deal with physical addresses
- Each process has its own mapping from virtual to physical addresses
- Enables:
  - Code and data to be located anywhere in physical memory
  - Isolation/separation of code and data of different processes in physical memory (the system translates the virtual memory to physical memory addresses such that multiple processes dont interefere with eachother)
  - code and data sharing between multiple processes (different virtual addresses can be mapped to the same data)



Virtual address space is divided into **Pages**
Physical address space is divided into **Frames**

A virtual page is mapped to a physical fram, if the page is in phyiscal memory other wise its mapped to a location in disk.

Demand Paging: If an accessed virtual page is not in memory, but on disk, the virtual memory system brings the page into a physical frame and adjusts the mapping

Page table: The table which stores the mapping of virtual pages to physical frames

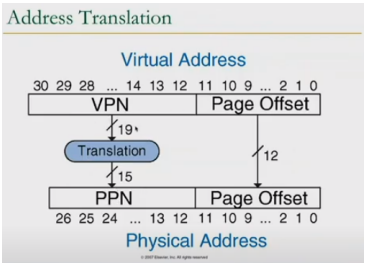
Hence physical memory is a cache for pages stored on disk. (in modern systems fully associative cache)

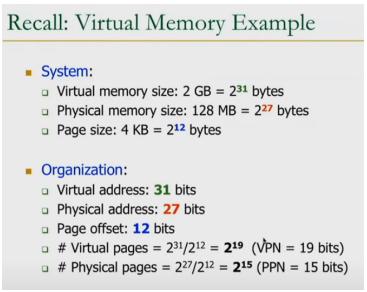
Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Page size: Amount of memory transferred from hard disk to DRAM (physical memory) at once (given by ISA)

Address translation: Determining the physical address from the virtual address

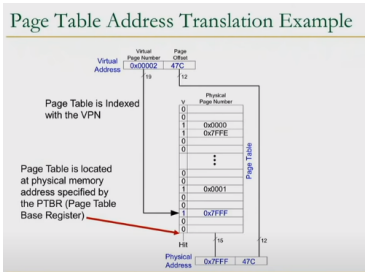
Goal: Most accesses hit in physical memory but the programs see the large capacity of virtual memory

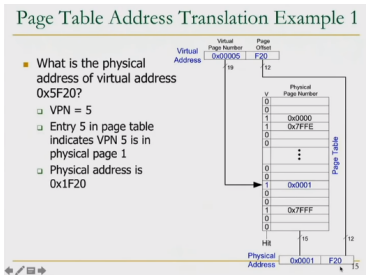


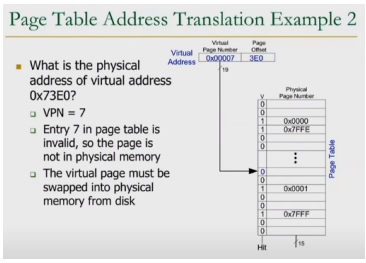


Address translation: The page table has an entry for each virtual page. Each page table entry has:

- **valid bit:** Whether the virtual page is located in physical memory (if not must be fetched from hard disk)
- **Physical page number:** where the virtual page is located in physical memory
- replacement policy
- dirty bits



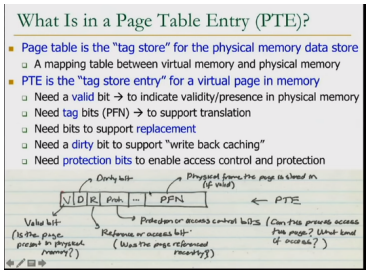


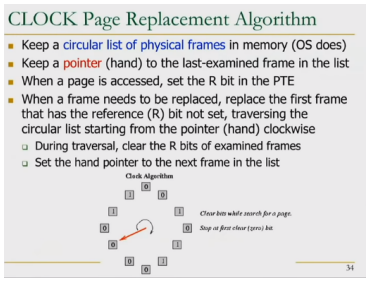


Page Table Challenges:

- Page table is large, atleast part of it needs to be located in physical memory. Solution: multi-level page tables.
- Each instruction fetch or load/store requires at least two memory accesses. One for address translation (page table read) and one to access data with the physical address (after translation)

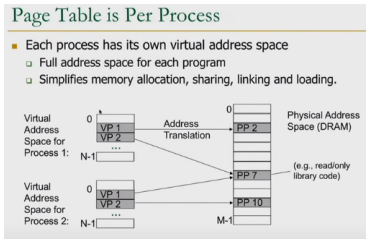
Translation lookaside buffer (TLB): Idea: Cache the page table entries (PTE's) in a hardware structure in the processor to speed up address translation. TLB is a small cache of most recently used translations (PTEs). This reduces the number of memory accesses required for most loads/stores to only one.





Page Replacement Algorithms:

Memory Protection: Multiple programs run at once. Each process has its own page table and can use entire virtual address space without worrying about where other programs are. A process can only access physical pages mapped in its page table and cannot overwrite memory of another process, this provides protection and isolation between processes.



Access Protection/Control via Virtual Memory Virtual memory allows page-level Access control i.e not every process is allowed to access every page. The idea is to store access control information on a page basis in the process's page table. We enforce access control at the same time as translation.