

ECE-425: Verilog & Processors

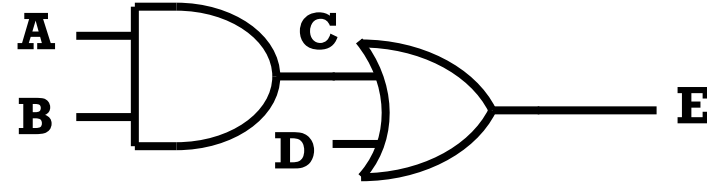
Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

Verilog Description of Combinational Circuits

```
1  assign #5 C = A && B;  
   assign #5 E = C || D;
```



```
2  assign #5 E = C || D;  
   assign #5 C = A && B;
```

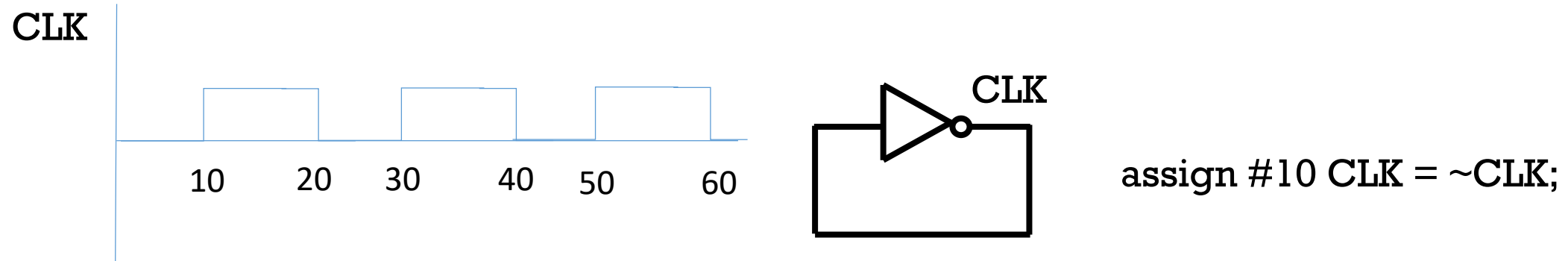
Case 1, 2:

Unlike a sequential program, the order of the above concurrent statements is unimportant.

In general, a signal assignment statement has the form

```
assign [#delay] signal_name = expression;
```

Verilog Description of Combinational Circuits



- In general, **Verilog is case sensitive**: that is, capital and lower-case letters are treated as different by the compiler and by the simulator.
- Signal names and other **Verilog identifiers** may contain letters, numbers, the underscore character, and it cannot start with a number or a \$ sign. The dollar sign (\$) is reserved as the first character for **system tasks**.
- Naming modules (**identifiers**)
 - **Valid**: adder, Mux_input, _error_code, Index_bit, etc
 - **Not Valid**: 4bitadder, \$error_code
- Every Verilog statement must be terminated with a semicolon. Spaces, tabs, and carriage returns are treated in the same way.
- Comment: `//` , `/*` code in the middle `*/`

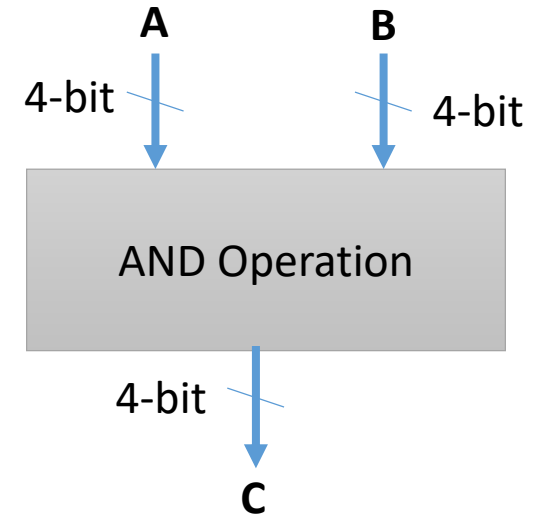
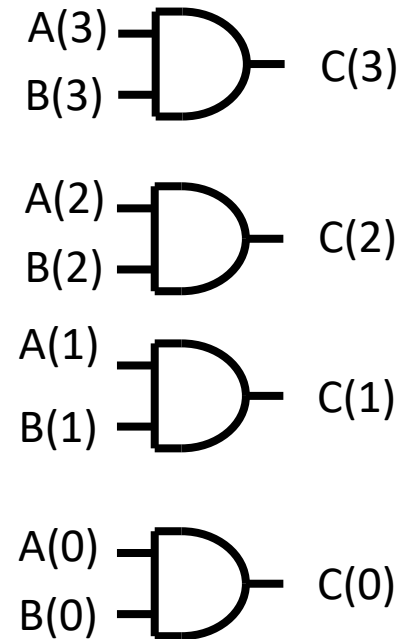
Verilog Description of Combinational Circuits

// Hard way (be tricky, if you needed somehow, use it)

```
assign C[3] = A[3] && B[3];  
assign C[2] = A[2] && B[3];  
assign C[1] = A[1] && B[3];  
assign C[0] = A[0] && B[3];
```

// Easy way assuming C, A and B are 4-bit vectors

```
assign C = A & B;
```



Full Adder (Verilog Continue)

Module FullAdder (X, Y, Cin, Cout, Sum);

Output Cout, Sum;

input X, Y, Cin;

assign Sum = X^Y^Cin;

assign Cout = (X && Y) || (X && Cin) || (Y && Cin);

endmodule



Sum = X xor Y xor Cin

Cout = X.Y xor Y.Cin xor X.Cin

4-bit Adder (Verilog Continue)

```
Module Adder4 (S, Co, A, B, Ci);
```

```
Output [3:0] S;
```

```
Output      Co;
```

```
input [3:0] A, B;
```

```
input      Ci;
```

```
wire [3:1] C; // is an internal signal
```

```
// Instantiate four copies of the FullAdder
```

```
FullAdder FA0 (A[0], B[0], ci,    C[1], S[0];
```

```
FullAdder FA1 (A[1], B[1], C[1], C[2], S[1];
```

```
FullAdder FA2 (A[2], B[2], C[2], C[3], S[2];
```

```
FullAdder FA3 (A[3], B[3], C[3], Co , S[3];
```

```
endmodule
```

Always(Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.

Module sequential_module (A, B, C, D, clk)

input clk;

output A,B,C, D;

reg A,B,C,D;

always @(posedge clk)

begin

A = B;

B = A;

end

always @(posedge clk)

begin

C <= D;

D <= C;

end

endmodule

Always block sensitivity list can be written as the following:

Always @(*)

begin

end

Modeling Flip-Flops (Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.

```
always @ (posedge CLK)
```

```
begin
```

```
    Q <= D;
```

```
end
```



D-FF

```
always @ (G or D)
```

```
begin
```

```
    if (G)
```

```
        Q <= D;
```

```
    end
```

```
end
```



Latch

```
always @ (posedge CLK or negedge ClrN )
```

```
begin
```

```
    if (~ClrN)
```

```
        Q <= 0;
```

```
    else
```

```
        Q <= D;
```

```
end
```



D-FF

Case statement (Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.

Case expression

choice1: sequential statement1

choice2: sequential statement2

.....

[default: sequential statements]

endcase;



Case Sel

2'b00: sequential statement1

2'b01: sequential statement2

2'b10: sequential statement2

2'b11: sequential statement2

default:

endcase;

Case Sel

2'b00: F = I0;

2'b01: F = I1;

2'b10: F = I2;

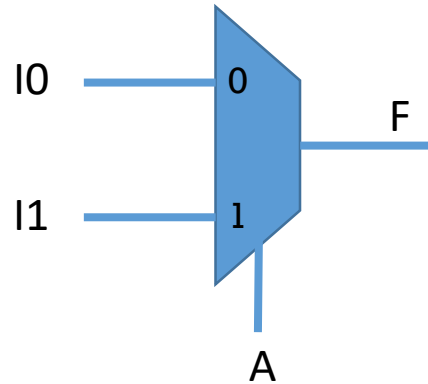
2'b11: F = I3;

default:

endcase;

Multiplexers(Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.



```
assign signal_name = condition ? Expression_T: Expression_F;
```

```
assign F= (A) ? I1: I0;
```

Always @(Sel or I0 or I1 or I2 or I3)

Case **Sel**

2'b00: F = I0;

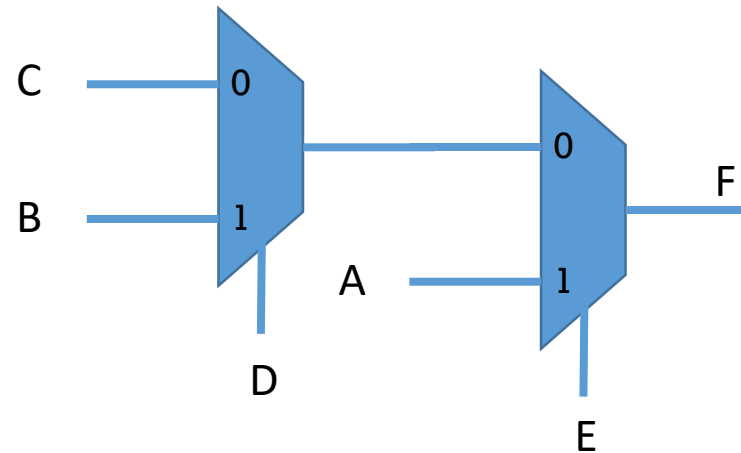
2'b01: F = I1;

2'b10: F = I2;

2'b11: F = I3;

default:

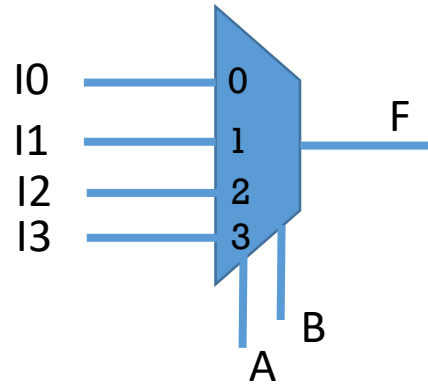
endcase;



```
assign F= E ? A: (D? B:C);
```

Multiplexers 4x1 (Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.



1
`assign F= (~A && ~B && I0) || (~A && B && I1) || (A && ~B && I2) || (A && B && I3);`

2
`assign F= (A) ? (B ? I3: I2) : (B ? I1 : I0);`

Latches are not comb hardware, it creates a variety of timing problems

4

`Always @(Sel or I0 or I1 or I2 or I3)`

`F = 0;`

`begin`

`if (Sel == 2'b00) F = I0;`

`else if (Sel == 2'b01) F = I1;`

`else if (Sel == 2'b10) F = I2;`

`else if (Sel == 2'b11) F = I3;`

`end`

Another way, by initializing at the beginning of the always statement

3 `Always @(Sel or I0 or I1 or I2 or I3)`

`Case Sel`

`2'b00: F = I0;`

`2'b01: F = I1;`

`2'b10: F = I2;`

`2'b11: F = I3;`

`default:`

`endcase;`

Can be removed
in Verilog if you
don't want to use

For if else and case statements, it is important to have all cases specified (to avoid latch in the translation process)

Muxs(Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.

// Behavioral Model of a 2 to 1 MUX (16-bit inputs)

```
module mux_2to1(Y, A, B, sel);  
    output [15:0] Y;  
    input [15:0] A, B;  
    input sel;  
    reg [15:0] Y;  
    always @(A or B or sel)  
        if (sel == 1'b0) Y = A;  
        else Y = B;  
endmodule
```

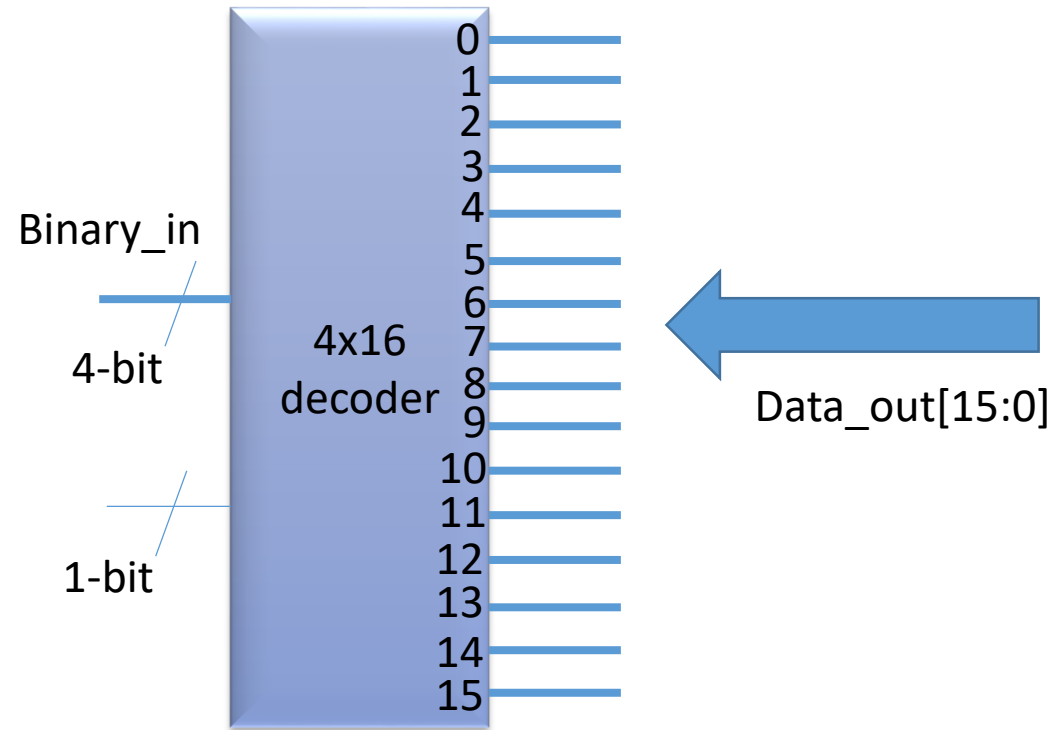
//Behavioral Model of a 4 to 1 MUX (16 data inputs)

```
module mux_4to1(Y, A, B, C, D, sel);  
    output [15:0] Y;  
    input [15:0] A, B, C, D;  
    input [1:0] sel;  
    reg [15:0] Y;  
    always @(A or B or C or D or sel)  
        case ( sel )  
            2'b00: Y = A;  
            2'b01: Y = B;  
            2'b10: Y = C;  
            2'b11: Y = D;  
            default: Y = 16'hxxxx;  
        endcase  
endmodule
```

Decoders (Verilog Continue)

Always block: can be used for building both sequential and combinational circuits.

```
Module decoder_using_case (  
    binary_in,  
    decoder_out,  
    enable  
);  
  
input [3:0] binary_in;  
input enable;  
output [15:0] decoder_out;  
reg [15:0] decoder_out;  
  
Always @ (enable or binary_in)  
begin  
    decoder_out = 0;  
    If (enable) begin  
        case (binary_in)  
            4'h0: decoder_out = 16'h0001;  
            .....  
        endcase  
    end  
end  
end  
endmodule
```



Decoder 2x4 (Verilog Continue)

/* Module of a 2 to 4 Decoder with an active high enable input and active low outputs. This model uses behavioral modeling via the "case" statement */

```
module decoder_2to4(Y3, Y2, Y1, Y0, A, B, en);  
    output Y3, Y2, Y1, Y0;  
    input A, B;  
    input en;  
    reg Y3, Y2, Y1, Y0;  
    always @(A or B or en)  
    begin  
        if (en == 1'b1)  
            case ( {A,B} )  
                2'b00: {Y3,Y2,Y1,Y0} = 4'b1110;  
                2'b01: {Y3,Y2,Y1,Y0} = 4'b1101;  
                2'b10: {Y3,Y2,Y1,Y0} = 4'b1011;  
                2'b11: {Y3,Y2,Y1,Y0} = 4'b0111;  
                default: {Y3,Y2,Y1,Y0} = 4'bxxxx;  
            endcase  
        if (en == 0){Y3,Y2,Y1,Y0} = 4'b1111;  
    end  
endmodule
```

Decoder 3x8 (Verilog Continue)

```
/* Module of a 3 to 8 Decoder with an active high enable input and active low outputs. This model uses a trinary continuous assignment statement for the combinational logic */
```

[illegible]

Processor performance

Example of how to calculate CPI:

The table below indicates frequency of all instruction types executed in a “typical” program and, from the reference manual, we are provided with a number of cycles per instruction for each type.

Instruction type	Frequency	Cycles
ALU instruction	50%	4
Load Instruction	30%	5
Store instruction	5%	4
Branch Instruction	15%	2

$$\text{CPI} = .5 * 4 + 0.3 * 5 + 0.05 * 4 + 0.15 * 2 = 4 \text{ cycles/instruction}$$

Processor performance

Example (1): CPU Time:

Consider an implementation of MIPS ISA with 500 MHz clock and

- Each ALU instruction takes 3 clock cycles
- Each branch/jump instruction takes 2 clock cycles
- Each sw instruction takes 4 clock cycles
- Each lw instruction takes 5 clock cycles

Also, consider a program that during its execution executes:

- X = 200 million ALU instructions
- Y = 55 million branch/jump instructions
- Z = 25 million sw instructions
- W = 20 million lw instructions

Find CPU time ?????????????????????????????????

Assume sequentially executing CPU

Processor performance

CPU Time:

Approach 1:

- $\text{Clock_cycles_for_a_program} = (x*3+y*2+z*4+w*5) = 910 * 10^6$ clock cycles
- $\text{CPU_time} = \text{clock_cycles_for_a_program} / \text{clock_rate} = 910 * 10^6 / 500 * 10^6 = 1.82$ seconds

Approach 2:

- $\text{CPI} = \text{clock_cycles_for_a_program} / \text{instruction_count}$
- $\text{CPI} = (x*3+y*2+z*4+w*5) / (x+y+z+w) = 3.03$ clock cycles/instruction
- $\text{CPU_time} = \text{instruction_count} * \text{CPI} / \text{clock_rate} = (x+y+z+w) * 3.03 / 500 * 10^6 = 300 * 10^6 * 3.03 / 500 * 10^6 = 1.82$ seconds

Processor performance

Example (2): CPU Time:

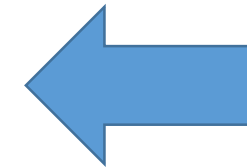
Consider an implementation of MIPS ISA with 1 GHz clock and

- Each ALU instruction takes 4 clock cycles
- Each branch/jump instruction takes 3 clock cycles
- Each sw instruction takes 5 clock cycles
- Each lw instruction takes 6 clock cycles

Also, consider a program that during its execution executes:

- X = 200 million ALU instructions
- Y = 55 million branch/jump instructions
- Z = 25 million sw instructions
- W = 20 million lw instructions

Same as the first example



Find CPI and CPU time ?????????????????????????????????

Assume sequentially executing CPU

Processor performance

CPI

$$\text{CPI} = (x*4 + y * 3 + z * 5 + w * 6) / (x + y + z + w) = 4.03 \text{ clock cycles/instruction}$$

CPU_time

$$\begin{aligned} \text{CPU_time} &= \text{CPU time} = \text{Instruction count} * \text{CPI} / \text{Clock rate} = \\ (x+y+z+w) * 4.03 / 1000 * 10^6 &= 300 * 10^6 * 4.03 / 1000 * 10^6 = 1.21 \text{ sec} \end{aligned}$$

Processor performance

Example 3

Suppose that when Program A is run, the user CPU time is 3 seconds, the elapsed wallclock time is 4 seconds, and the system performance is 10 MFLOP/sec. Assume that there are no other processes taking any significant amount of time, and the computer is either doing calculations in the CPU, or doing I/O, but it can't do both at the same time. We now replace the processor with one that runs **six times faster**, but doesn't affect the I/O speed. What will the user CPU time, the wallclock time, and the MFLOP/sec performance be now?

$$\text{CPU performance}_B / \text{CPU performance}_A = \text{CPU time}_A / \text{CPU time}_B$$

$$6 = 3 / \text{CPU time}_B$$

$$\text{User CPU Time} = 0.5 \text{ seconds}$$

Since the I/O time is unaffected by the performance increase, it still takes 1 second to do I/O.

Therefore it takes $1 + 0.5 = 1.5$ seconds to run Program A on the faster CPU

$$\text{Wallclock Time} = 1.5 \text{ seconds}$$

$$\text{System Performance in MFLOPS} = \text{Number of Floating Point Operations} * 10^6 / \text{Wallclock Time}$$

$$\text{Old System Performance (10)} = \text{\#FLOP} * 10^6 / 4$$

$$\text{\#FLOP} = 40 * 10^6$$

$$\text{New System Performance} = 40 * 10^6 / 1.5$$

$$\text{MFLOP/sec} = 26.667$$

Processor performance

Example 4

You are on the design team for a new processor. The clock of the processor runs at 200 MHz. The following table gives instruction frequencies for Benchmark B, as well as how many cycles the instructions take, for the different classes of instructions. For this problem, we assume that (unlike many of today's computers) the processor only executes one instruction at a time.

Instruction Type	Frequency	Cycles
Loads & Stores	30%	6 cycles
Arithmetic Instructions	50%	4 cycles
All Others	20%	3 cycles

Calculate the CPI for Benchmark B.

Processor performance

Solution:

If we assume that there are 100 instructions, then:

- 30 of them will be loads and stores.
- 50 of them will be arithmetic instructions
- 20 of them will be all others.
 - $(30 * 6) + (50 * 4) + (20 * 3) = 440$ cycles/100 instructions

Therefore, there are 4.4 Cycles per instruction.



The CPU execution time on the benchmark is exactly 11 seconds. What is the "native MIPS" processor speed for the benchmark in millions of instructions per second?

The formula for calculating MIPS is:

$$\text{MIPS} = \text{Clock rate} / (\text{CPI} * 10^6)$$

The clock rate is 200MHz so...

$$\text{MIPS} = (200 * 10^6) / (4.4 * 10^6) = 45.454545$$

Processor performance

Solution (cont):

The hardware expert says that if you double the number of registers, the cycle time must be increased by 20%. What would the new clock speed be (in MHz)?

Clock time = $1/\text{Cycle Time}$

Cycle Time = $1/\text{Clock Time}$

Cycle Time = $1/(200 * 10^6) = 5 * 10^{-9}$

The cycle time is then increased by 20%:

$(5 * 10^{-9}) * 1.2 = 6 * 10^{-9}$

The new clock rate is thus:

$1/(6 * 10^{-9}) = 166.667 * 10^6$ or 166.667 MHz

Processor performance

Solution(cont):

The compiler expert says that if you double the number of registers, then the compiler will generate code that requires only half the number of Loads & Stores. What would the new CPI be on the benchmark?

There were 100 instructions in part b, so we will reduce the number of loads and stores by half, and this will reduce the total number of instructions. So the new instruction mix will be:

- 15 Loads and Stores

- 50 Arithmetic Instructions

- 20 All Others

The total number of instructions is now 85, so the answer is:

$$((15 * 6) + (50 * 4) + (20 * 3)) / 85 = 350 \text{ cycles} / 85 \text{ instructions} = 4.12 \text{ CPI}$$

Processor performance

Solution(cont):

How many CPU seconds will the benchmark take if we double the number of registers (taking into account both changes described above)?

CPU seconds = (Number of instructions * Number of Clocks per instructions)/Clock Rate

First thing we need to do, is calculate the number of instructions which execute in 11 seconds on the new benchmark - the one with half the number of loads and stores.

To do this, we will need to figure out how many instructions execute on the original benchmark in 11 seconds. Since we know the MIPS or how many *Millions of Instructions Per Second* for the original benchmark, we say:

$$(45.45 * 10^6) * 11 = 500 * 10^6 \text{ instructions in 11 seconds}$$

Processor performance

Solution (cont):

Now we need to figure out how many of those are Loads and Stores so:

$(500 * 10^6) * .3 = 150 * 10^6$ are Load and Store instructions because the chart says that 30% of all instructions are Loads and Stores.

Now we need to cut this number in half, because the new benchmark says that we have half the number of loads and stores , but the cycle time increases by 20%.

Therefore there are only $75 * 10^6$ loads and stores. This also means that there are now less total instructions, $425 * 10^6$ total instructions.

The final solution is:

$$((425 * 10^6) * 4.12) / (166.667 * 10^6) = 10.548 \text{ seconds}$$

MIPS programming

Example:

Here is the load word instruction in assembly language:

```
lw d, off(b)    # $d ← word from memory address b+off  
                # b is a register. Off is 16-bit two's complement  
                # (the data from memory is available in $d after  
                # a one machine cycle delay
```

At the execution time two things happen: 1) an address is calculated by adding the base register b with the offset off, and 2) data is fetched from memory at that address

Because it takes time to copy data from memory, it takes two machine cycles before the data is available in register \$d. In terms of assembly language this means the instruction immediately a lw should not use \$d

MIPS programming

Question:

Write the instruction that loads the word at address 0x00400060 into register \$8. Assume that register \$10 contains 0x00400000

Answer:

0x00400060 --- address of data

0x00400000 --- address in \$10


\$8 ---- destination register

The instruction is:

lw \$8, 0x60(\$10)

Here is the machine code version of the instruction. It specifies the base register, the destination register, and the offset.

- The 32-bit address in \$10 is: 0x00400000
- The offset is sign-extended to 32-bit: 0x00000060
- The memory address is the 32-bit sum of the above: 0x00400060
- Main memory is asked for data from that address
- After a one machine cycle delay, the data reaches \$8. → the 4bytes



100011	01010	01000	0000	0000	0110	0000	-- fields of the instruction
lw	\$10	\$8	0	0	6	0	
opcode	base	dest		offset			-- meaning of the fields
lw	\$8, 0x60(\$10)						-- assembly language