CS61C : Machine Structures

Lecture 5.1.1

CPU Design I

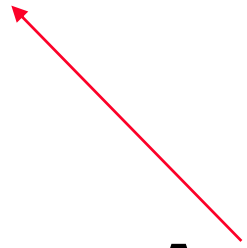
2004-07-19

Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

Review: Verilog Dataflow Paradigm

```
module and_or(Z, A, B, Sel);
 input A, B, Sel
 output Z;
 wire Z;
 assign #5 Z = Sel ? (A & B) : (A | B);
endmodule
```

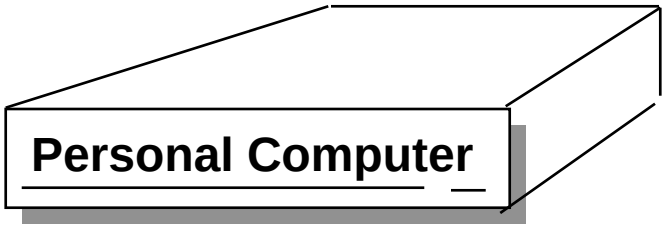


- Continuous Assignment:
  - Limited to simple operations
    - Simple translation to structural
    - Trinary if (? :), logic operators
  - Excellent for CL.

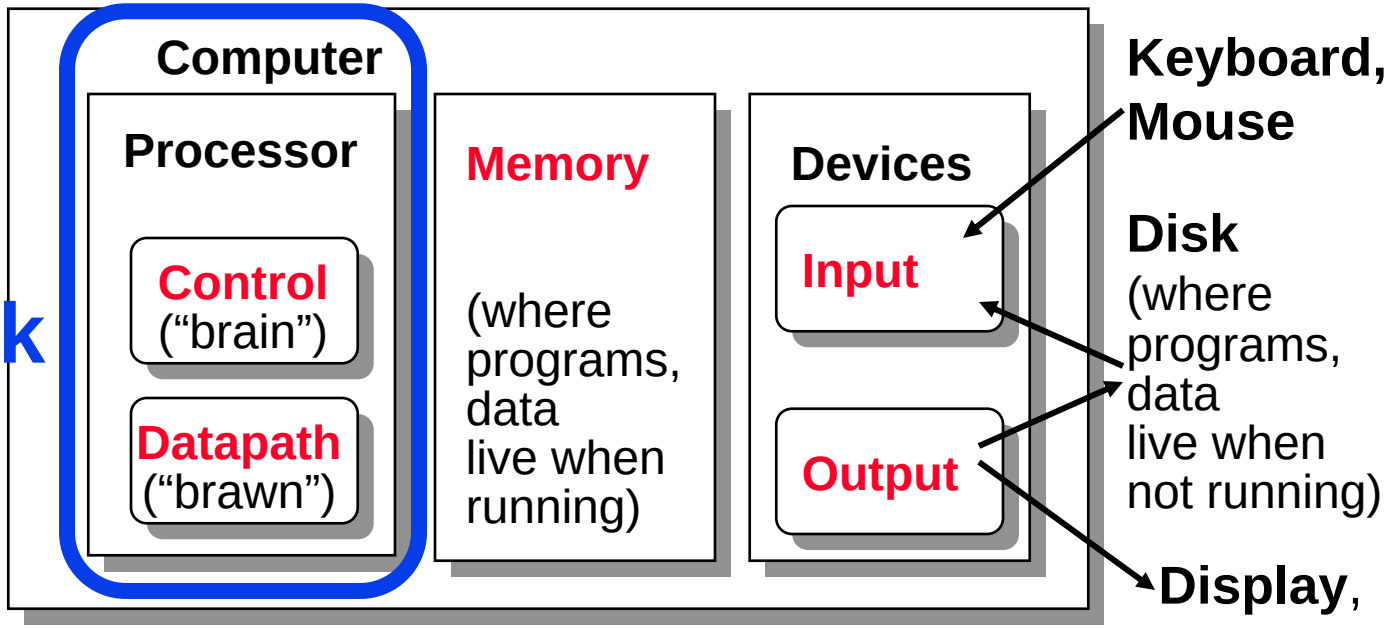
Review: What's a reg??

- Output types:
  - Structural → Wire
    - wire a,b,c; and (a, b, c);
  - Continuous Assign → Wire
    - wire a,b,c; assign a = b & c;
  - Procedural Assign → REG
    - wire b,c; reg a; always @ (b or c) a = b & c;
- Why?
  - Structural, continuous always function of current input. → simple wire
  - REG ←→ Verilog has to remember value

Anatomy: 5 components of any Computer



This week



Outline

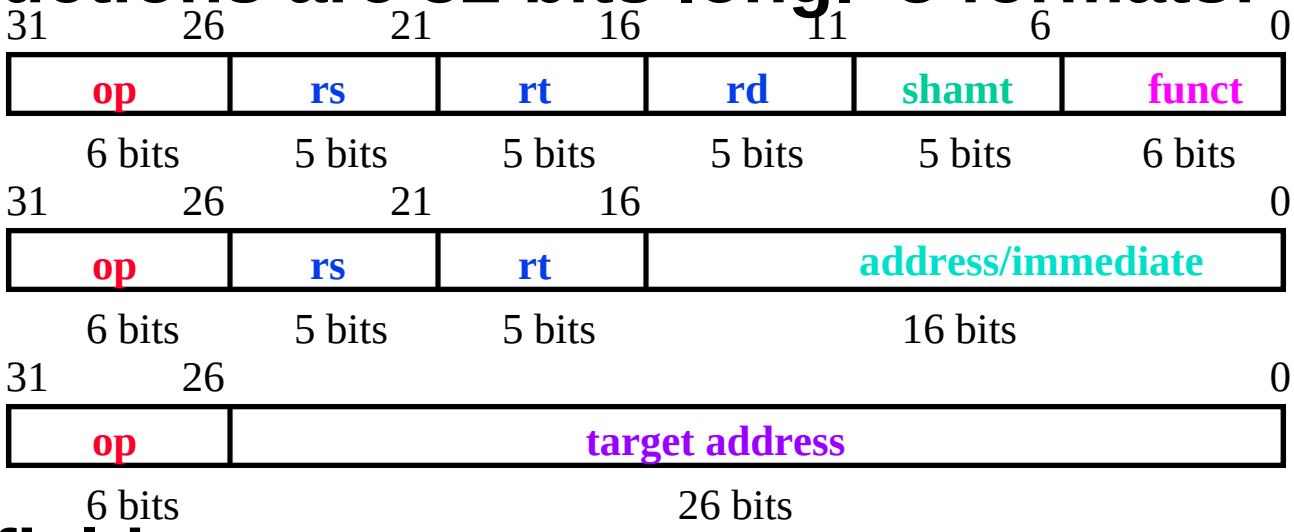
- Design a processor: step-by-step
- Requirements of the Instruction Set
- Hardware components that match the instruction set requirements

How to Design a Processor: step-by-step

- 1. Analyze instruction set architecture (ISA) => datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

Step 1: The MIPS Instruction Formats

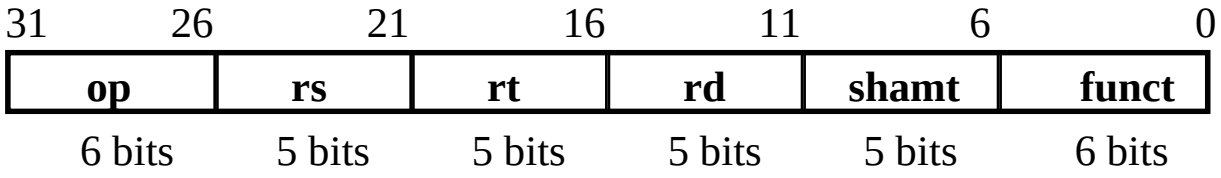
- All MIPS instructions are 32 bits long. 3 formats:
  - R-type
  - I-type
  - J-type



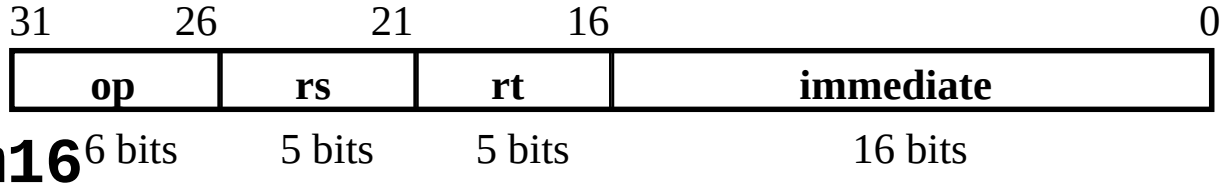
- The different fields are:
  - op: operation ("opcode") of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field
  - address / immediate: address offset or immediate value
  - target address: target address of jump instruction

Step 1: The MIPS-lite Subset for today

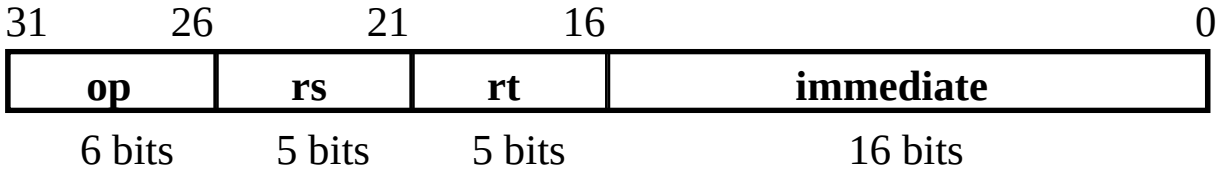
- **ADD and SUB**
  - `addU rd,rs,rt`
  - `subU rd,rs,rt`



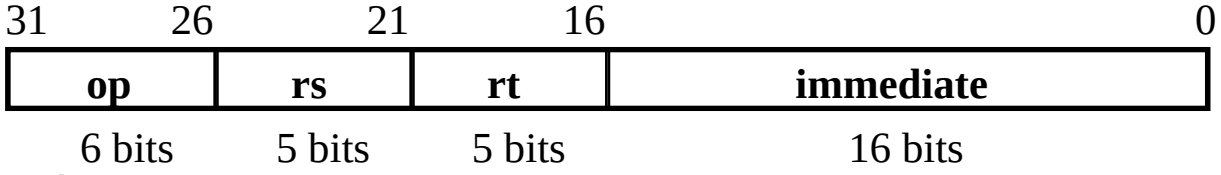
- **OR Immediate:**
  - `ori rt,rs,imm16`



- **LOAD and STORE Word**
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`



- **BRANCH:**
  - `beq rs,rt,imm16`



Step 1: Register Transfer Language

- **RTL gives the meaning of the instructions**

{op , rs , rt , rd , shamt , funct} = MEM[PC]

{op , rs , rt , Imm16} = MEM[PC]

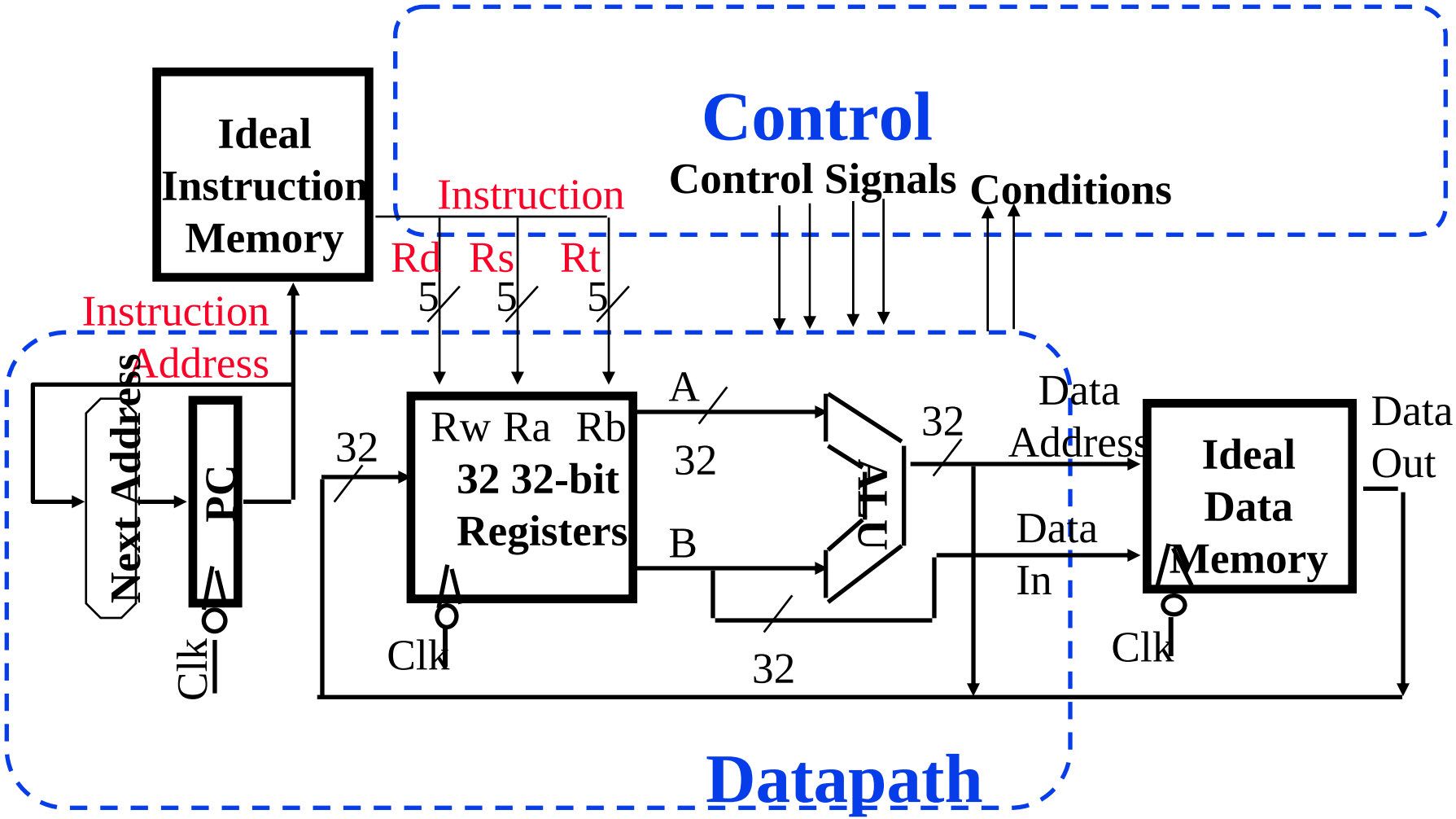
- **All start by fetching the instruction**

inst	Register Transfers	
ADDU	R[rd] = R[rs] + R[rt];	PC = PC + 4
SUBU	R[rd] = R[rs] – R[rt];	PC = PC + 4
ORI	R[rt] = R[rs] | zero_ext(Imm16);	PC = PC + 4
LOAD	R[rt] = MEM[R[rs] + sign_ext(Imm16)];	PC = PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] = R[rt];	PC = PC + 4
BEQ	if (R[rs] == R[rt]) then PC = PC + 4 + sign_ext(Imm16)] << 2 else PC = PC + 4	

Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - instructions & data
- Registers (R: 32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender (sign extend)
- Add and Sub register or extended immediate
- Add 4 or extended immediate to PC

Step 1: Abstract Implementation



How to Design a Processor: step-by-step

- 1. Analyze instruction set architecture (ISA) => datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic (hard part!)

Step 2a: Components of the Datapath

- **Combinational Elements**
- **Storage Elements**
  - **Clocking methodology**

Combinational Logic: 16-bit Sign Extender

```
//Sign extender from 16- to 32-bits.
module signExtend (in,out);
   input  [15:0] in;
   output [31:0] out;
   reg    [31:0] out;

 assign
 out = { in[15], in[15], in[15], in[15],
         in[15], in[15], in[15], in[15],
         in[15], in[15], in[15], in[15],
         in[15], in[15], in[15], in[15],
         in[15:0] };
endmodule // signExtend
```

Combinational Logic: 2-bit Left Shift

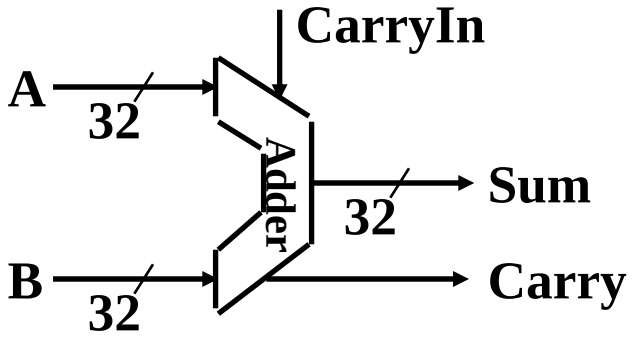
```
// 32-bit Shift left by 2
module leftShift2 (in,out);
    input [31:0] in;
    output [31:0] out;

    assign
    out = { in[29:0], 1'b0, 1'b0 };
endmodule // leftShift2
```

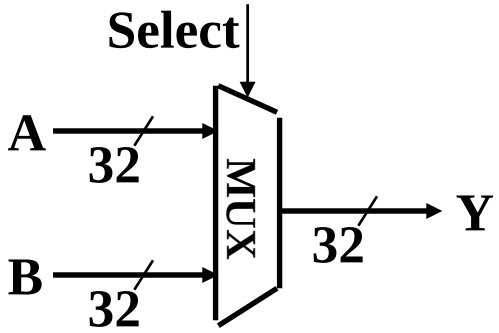
(Also: assign out = in[29:0] << 2;)

Combinational Logic: More Elements

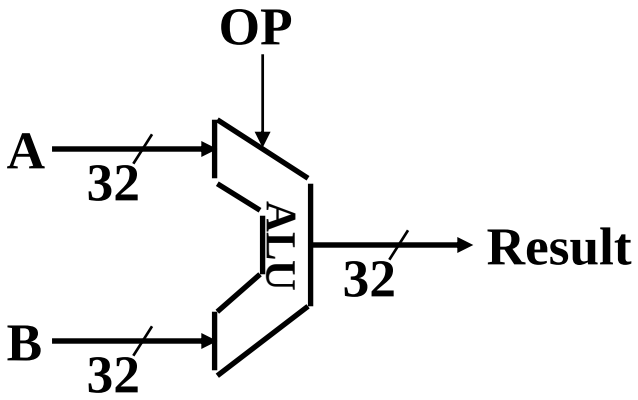
- Adder



- MUX



- ALU



Combinational Logic: 32-bit Adder

```
//Behavioral model of 32-bit adder.
module add32 (S,A,B);
   input  [31:0] A,B;  // 32 bits
   output [31:0] S;
   reg    [31:0] S;

   always @ (A or B)
     S = A + B;
endmodule // add32
```

How do we make this dataflow?

Combinational Logic: 32-bit Mux

```
// Behavioral model of 32-bit wide
// 2-to-1 multiplexor.
module mux32 (in0,in1,select,out);
   input [31:0] in0,in1;
   input        select;
   output [31:0] out;

   reg [31:0] out;
   always @ (in0 or in1 or select)
     if (select) out=in1;
     else out=in0;

endmodule // mux32
```

CL: ALU for MIPS-lite (1/4)

- Addition, subtraction, logical OR, ==:

```
ADDU  R[rd] = R[rs] + R[rt]; ...

SUBU  R[rd] = R[rs] – R[rt]; ...

ORI  R[rt] = R[rs] | zero_ext(Imm16)...

BEQ   if ( R[rs] == R[rt] )...
```

- Test to see if output == 0 for any ALU operation gives == test. How?
- P&H Section 4.5 also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- Behavioral ALU follows sec 4.5

CL: ALU (2/4)

```
//Behavioral model of ALU:
// 8 functions and "zero" flag,
// A is top input, B is bottom,
// according to P&H figure 5.19.

module ALU (A,B,control,zero,result);
    input  [31:0] A, B;
    input  [2:0]  control;
    output zero; // used for beq,bne
    output [31:0] result;


    reg [31:0] result, temp;
    always @ (A or B or control)...
```

CL: ALU (3/4)

```
reg [31:0]       result, C;
always @ (A or B or control)
 begin
 case (control)
   3'b000: // AND
     result=A&B;
   3'b001: // OR
     result=A|B;
   3'b010: // add
     result=A+B;
   3'b110: // subtract
     result=A-B;
   3'b111: // slt
     result=(A<B)? 1 : 0;
    endcase // case(control)
 end // always @ (A or B or control)
 assign zero = (result==0);
endmodule // ALU
```

☹Unsigned compare! Can you fix it?

CL: ALU (4/4)

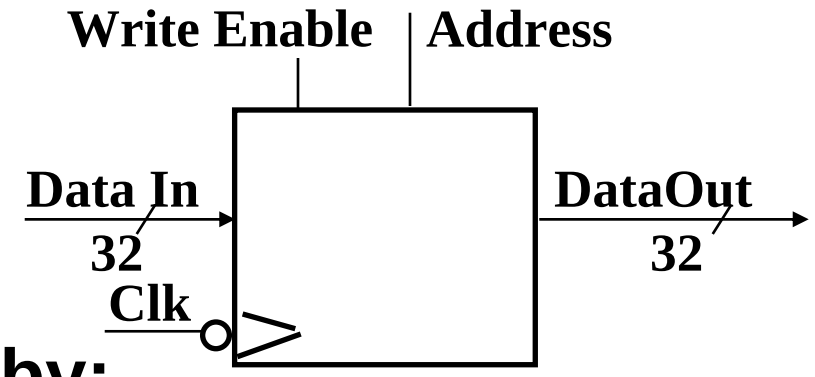
```
// if A and B have the same sign,
// then A<B works(slt == 1 if A-B<0)
// if A and B have different signs,
// then A<B if A is negative
// (slt == 1 if A<0)

   ...
   3'b111: // slt
      begin
        temp = A - B;
        result = (A[31]^B[31]) ?
                          A[31] :
                    temp[31];
      end
    ...
```

Step 2b: Components of the Datapath

- **Combinational Elements**
- **Storage Elements**
  - Clocking methodology

Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

Verilog Memory for MIPS Interpreter (1/3)

```
//Behavioral modelof Random Access Memory:
// 32-bit wide, 256 words deep,
// asynchronous read-port if RD=1,
// synchronous write-port if WR=1,
// initialize from hex file ("data.dat")
// on positive edge of reset signal,
// dump to binary file ("dump.dat")
// on positive edge of dump signal.
module mem
 (CLK,RST,DMP,WR,RD,address,writeD,readD);
    input CLK, RST, DMP, WR, RD;
    input [31:0] address, writeD;
    output [31:0] readD;
    reg [31:0] readD;
    parameter  memSize=256;
    reg [31:0] memArray [0:memSize-1];
    integer    chann,i;
```

Verilog Memory for MIPS Interpreter (2/3)

```
integer    chann,i;
always @ (posedge RST)
  $readmemh("data.dat", memArray);

// write if WR & positive clock edge (synchronous)
always @ (posedge CLK)
  if (WR) memArray[address[9:2]] =
                              writeD;

// read if RD, independent of clock (asynchronous)
always @ (address or RD)*
  if (RD)
      readD = memArray[address[9:2]];

endmodule
```

☹See how sneaky sensitivity lists can be!

Use an assign!

Why is it "memArray[address[9:2]]"?

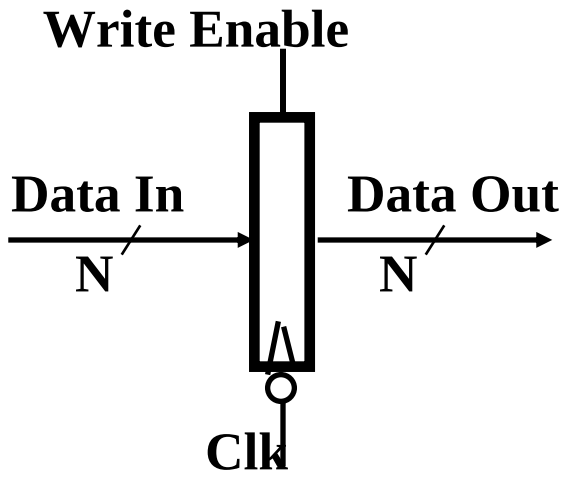
- Our memory is always byte-addressed
  - We can `lb` from 0x0, 0x1, 0x2, 0x3, …
- `lw` only reads word-aligned requests
  - We only call `lw` with 0x0, 0x4, 0x8, 0xC, …
  - I.e., the last two bits are always 0
- memArray is a word wide and 2^8 deep
  - `reg [31:0] memArray [0:256-1];`
  - Size = 4 Bytes/row * 256 rows = 1024 B
  - If we're simulating `lw`/`sw`, we R/W words
  - What bits select the first 256 words? [9:2]!
  - 1st word = 0x0 = 0b000 = memArray[0];
    2nd word = 0x4 = 0b100 = memArray[1], etc.

Verilog Memory for MIPS Interpreter (3/3)

```
  end;
  always @ (posedge DMP)
    begin
      chann = $fopen("dump.dat");
      if (chann==0)
        begin
          $display("$fopen of dump.dat
failed.");
          $finish;
        end       // Temp variables chan, i
      for (i=0; i<memSize; i=i+1)
        begin
          $fdisplay(chann, "%b",
                            memArray[i]);
        end
    end // always @ (posedge DMP)
endmodule // mem
```

Storage Element: Register (Building Block)

- 32-bit Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input (CE)
  - Write Enable:
    - negated (or deasserted) (0): Data Out will not change
    - asserted (1): Data Out will become Data In



Verilog 32-bit Register

```
// Behavioral model of 32-bit Register:
// positive edge-triggered,
// synchronous active-high reset.
module reg32 (CLK,Q,D,RST);
    input  [31:0] D;
    input  CLK, RST;
    output [31:0] Q;

    reg [31:0] Q;
    always @ (posedge CLK)
      if (RST) Q = 0; else Q = D;
endmodule // reg32
```