All MIPS instructions are 32 bits long. 3 formats:

1. R-type
   - 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

2. I-type
   - op: operation ("opcode") of the instruction
   - rs, rt: source register specifiers
   - imm16: 16-bit immediate value

3. J-type
   - op: operation ("opcode") of the instruction
   - rt: source register specifier
   - imm16: 16-bit target address

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

ADDU and SUBU
- 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 1a: The MIPS-lite Subset for today

1. Analyze instruction set architecture (ISA) \( \Rightarrow \) 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
Register Transfer Language (Behavioral)

• RTL gives the meaning of the instructions
  \begin{align*}
  \{ \text{op, rs, rt, rd, shamt, funct} \} &= \text{MEM}[\text{PC}] \\
  \{ \text{op, rs, rt, Imm16} \} &= \text{MEM}[\text{PC}]
  \end{align*}

• All start by fetching the instruction
  \begin{align*}
  \text{inst} = \text{MEM}[\text{PC}]
  \end{align*}

• \text{ADDU} \quad \text{R[rd]} = \text{R[rs]} + \text{R[rt]}; \quad \text{PC} = \text{PC} + 4

• \text{SUBU} \quad \text{R[rd]} = \text{R[rs]} - \text{R[rt]}; \quad \text{PC} = \text{PC} + 4

• \text{ORI} \quad \text{R[rt]} = \text{R[rs]} \vert \text{zero_ext(Imm16)}; \quad \text{PC} = \text{PC} + 4

• \text{LOAD} \quad \text{R[rt]} = \text{MEM}[\text{R[rs]} + \text{sign_ext(Imm16)}]; \quad \text{PC} = \text{PC} + 4

• \text{STORE} \quad \text{MEM}[\text{R[rs]} + \text{sign_ext(Imm16)}] = \text{R[rt]}; \quad \text{PC} = \text{PC} + 4

• \text{BEQ} \quad \text{if} (\text{R[rs]} == \text{R[rt]}) \text{ then}
  \begin{align*}
  \text{PC} = \text{PC} + 4 + (\text{sign_ext(Imm16)} || 00)
  \end{align*}
  \quad \text{else } \text{PC} = \text{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 2: Components of the Datapath

• Combinational Elements
• Storage Elements
  - Clocking methodology

Step 2.1: Components of the Datapath

16-bit Sign Extender for MIPS Interpreter

```verilog
module signExtend (in, out);
  input [15:0] in;
  output [31:0] out;
  reg [31:0] out;
  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], in[15], in[15], in[15] };
endmodule // signExtend
```

2-bit Left Shift for MIPS Interpreter

```verilog
module leftShift2 (in, out);
  input [31:0] in;
  output [31:0] out;
  reg [31:0] out;
  out = { in[29:0], 1'b0, 1'b0 };
endmodule // leftShift2
```

Combinational Logic Elements (Building Blocks)

- **Adder**

- **MUX**

- **ALU**
Verilog 32-bit Adder for MIPS Interpreter

// Behavioral model of 32-bit adder.
module add32 (S,A,B);
    input [31:0] A,B;
    output [31:0] S;
    reg [31:0] S;
    always @ (A or B)
        S = A + B;
endmodule // add32

Verilog 32-bit Register for MIPS Interpreter

// 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

ALU Needs for MIPS-lite + Rest of MIPS

* 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 also adds AND,
  * Set Less Than (1 if A < B, 0 otherwise)

Behavioral ALU follows chap 5

Verilog ALU for MIPS Interpreter (1/3)

// Behavioral model of ALU:
// 8 functions and "zero" flag,
// A is top input, B is bottom
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 zero;
    reg [31:0] result, C;
    always @ (A or B or control)...
endmodule // ALU

Verilog ALU for MIPS Interpreter (2/3)

reg [31:0] result, C;
always @ (A or B or control) begin
    case (control)
        3'b000: // AND
            result=A&B;
            zero=1'b0;
            endcase // case(control)
        3'b001: // OR
            result=A|B;
            zero=1'b0;
        3'b010: // add
            result=A+B;
        3'b100: // subtract
            result=A-B; // Documents bugs below
        3'b101: // set on less than
               // old version (fails if A is negative and B is positive)
            result = (A<B)? 1 : 0;
            // Why did it fail?
        3'b110: // set on less than
               // new version (correct)
            result = (A<B)? 1 : 0;
            zero = (result==0) ? 1'b1 : 1'b0;
        endcase // case(control)
endcase // case(control)

Verilog ALU for MIPS Interpreter (3/3)

reg [31:0] result, C;
always @ (A or B or control) begin
    if (control==3'b111)
        result = (A<B)? 1 : 0;
    if (control==3'b110)
        result = (A<B)? 1 : 0;
    if (control==3'b110)
        result = (A<B)? 1 : 0;
end // always @ (A or B or control)
Verilog Memory for MIPS Interpreter (1/3)

```verilog
// Behavioral model of 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; // ~ Constant dec.
  reg [31:0] memArray [0:memSize-1];
  integer chann,i;
begin
  // Temp variables: for loops
  for (i=0; i<memSize; i++)
    begin
      // write if WR & positive clock edge (synchronous)
      if (WR & posedge CLK)
        memArray [address[9:2]] = writeD;
      // read if RD, independent of clock (asynchronous)
      if (RD & posedge CLK)
        readD = memArray [address[9:2]];
      $display("%s",readD);
    end
$fdisplay(chann, "%h", memArray[i]);
endmodule
```

Verilog Memory for MIPS Interpreter (2/3)

```verilog
integer chann,i;
always @ (posedge RST) begin
  always @ (posedge CLK)
    if (WR) memArray[address[9:2]] = writeD;
    $readmemh("data.dat", memArray); // read from hex file
  always @ (address or RD)
    if (RD)
      begin
        readD = memArray[address[9:2]];
        $display("Getting address %h containing %h", address[9:2], readD);
      end
    // read if RD, independent of clock (asynchronous)
end
```

Verilog Memory for MIPS Interpreter (3/3)

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

Summary: Single cycle datapath

"5 steps to design a processor"

- Analyze instruction set => datapath requirements
- Select set of datapath components & establish clock methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- Assemble the control logic

"Control is the hard part.  "Next time!"

Peer Instruction

A. We should use the main ALU to compute PC=PC+4
B. We’re going to be able to read 2 registers and write a 3rd in 1 cycle
C. Datapath is hard, Control is easy

---

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.”

Datapath

- Datapath registers and write a 3
- We should use the main ALU
- Compute PC=PC+4
- Peer Instruction