Overview

- Instruction encoding
- Control unit design
  - for single cycle datapath
- Modeling design with VHDL*

Recall: MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
    - op: operation 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
  - I-type
    - op: operation of the instruction
    - rs, rt: the source and destination register specifiers
    - shamt: shift amount
    - immediate: address offset or immediate value
  - J-type
    - op: target address

Instruction Encoding

- Instruction encoding uses binary codes to represent operations and operands.
  - See MIPS reference data sheet in the textbook for MIPS instruction encoding
  - Example
    | Instr. | R-type | ori | lw | sw | beq | jump |
    |---|---|---|---|---|---|---|
    | op | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |

- Control unit design is closely related to instruction encoding
**Recall: Typical Steps of Processor Design**

1. Analyse instruction set and datapath requirements
   - the meaning of each instruction is given by the register transfer operations
     - Data from one storage location to another storage location
     - datapath must support each register transfer operation
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath to meet the requirements
4. Analyse implementation of each instruction to determine control points that affect the register transfer.
5. Assemble the control logic

**Step 4: Determine Control Signals**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>SUB</td>
<td>R[rd] ← R[rs] – R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>ORi</td>
<td>R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, ExtOp = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] ← MEM[ Rs + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, ExtOp = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[ Rs + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, ExtOp = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( R[rs] == R[rt] ) then PC ← PC + sign_ext(Imm16)</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = EQUAL, ALUsrc = RegB, ALUctr = “sub”</td>
</tr>
</tbody>
</table>
Logic for each control signal

- **nPC_sel**: if (OP == BEQ) then EQUAL else 0
- **ALUsrc**: if ((OP == “000000”)||(OP == BEQ)) then “regB” else “immed”
- **ALUctr**: if (OP == “000000”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- **ExtOp**: if (OP == ORi) then “zero” else “sign”
- **MemWr**: if (OP == Store)
- **MemtoReg**: if (OP == Load)
- **RegWr**: if ((OP == Store) || (OP == BEQ)) then 0
  else 1
- **RegDst**: if ((OP == Load) || (OP == ORi)) then 0
  else 1

### Summary of the Control Signals

![Summary of the Control Signals](image)

### Local Decoding of ALU Control Signal

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>RegDst</td>
<td>ALUsrc</td>
<td>MemtoReg</td>
<td>RegWrite</td>
<td>MemWrite</td>
<td>Branch</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;: “R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
<td></td>
</tr>
</tbody>
</table>

**Main Control**

- Local decoding makes for simpler, smaller, faster control components

### Encoding of ALUop

- For the MIPS subset we consider here, ALUop has to be 2 bits wide to represent:
  - (1) “R-type” instructions
  - “I-type” instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) “R-type” instructions
  - “I-type” instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)...

![Encoding of ALUop](image)
Decoding of the “func” Field

- **R-type**: ori lw sw beq jump
- **ALUop** (Symbolic)
- **ALUop<2:0>**: 1 00 0 10 0 00 0 00 0 01

<table>
<thead>
<tr>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 00</td>
<td>0 10</td>
<td>0 00</td>
<td>0 01</td>
<td>xxx</td>
</tr>
</tbody>
</table>

Truth Table for ALUctrl

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 00</td>
<td>0 10</td>
<td>0 00</td>
<td>0 01</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUop bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>func bit&lt;3&gt; bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>ALU Operation bit&lt;2&gt; ALU.ctrl bit&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>x x x x</td>
<td>Add 0 1 0</td>
</tr>
<tr>
<td>0 1 x</td>
<td>x x x x</td>
<td>Subtract 1 1 0</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 0 0 0 0</td>
<td>Add 0 1 0</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 0 1 0</td>
<td>Subtract 1 1 0</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 1 0 0</td>
<td>And 0 0 0</td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0</td>
<td>Or 0 0 1</td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0</td>
<td>Set on &lt; 1 1 1</td>
</tr>
</tbody>
</table>

ALUcontrol as defined in Ch 4

<table>
<thead>
<tr>
<th>func&lt;5:0&gt; Instruction Operation</th>
<th>ALUctrl&lt;2:0&gt; ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0000 add</td>
<td>000 And</td>
</tr>
<tr>
<td>10 0010 subtract</td>
<td>001 Or</td>
</tr>
<tr>
<td>10 0100 and</td>
<td>010 Add</td>
</tr>
<tr>
<td>10 0101 or</td>
<td>110 Subtract</td>
</tr>
<tr>
<td>10 1010 set-on-less-than</td>
<td>111 Set-on-less-than</td>
</tr>
</tbody>
</table>

Logic Equation for ALUctrl<2>

\[
\text{ALUctrl<2>} = \neg \text{ALUop<2>} \land \text{ALUop<0>} + \\
\text{ALUop<2>} \land \neg \text{func<2>} \land \text{func<1>} \land \neg \text{func<0>}
\]

Logic Equation for ALUctrl<1>

\[
\text{ALUctrl<1>} = \neg \text{ALUop<2>} \land \neg \text{ALUop<1>} + \\
\text{ALUop<2>} \land \neg \text{func<2>} \land \text{func<1>} \land \neg \text{func<0> (if func<3> + func<1>)}
\]
Logic Equation for ALUctr<0>

<table>
<thead>
<tr>
<th>ALUop</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
</tr>
<tr>
<td>0 1 x</td>
<td>x x x</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0</td>
</tr>
</tbody>
</table>

\[
ALUctr<0> = \overline{ALUop<2>} \land ALUop<1> + \overline{ALUop<2>} \land \overline{func<3>} \land func<2> \land \overline{func<1>} \land func<0> + \overline{ALUop<2>} \land func<3> \land \overline{func<2>} \land func<1> \land \overline{func<0>}
\]
Exercise 1:

- Given the processor design, highlight the components for three different types of instructions

R-Type Instruction
Exercise 2:

Implementing Jump Instruction

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode
VHDL

- A Hardware Description Language for very high speed integrated circuits
- Hardware systems have
  - special attributes
    - Delay and concurrency
  - Interactive relationship between components.
- Basic structure of VHDL code
  - Entity
    - For interface description
  - Architecture
    - For function description

**Entity**

- Specify input and output *signals* of the hardware components
  - Port clause is used.
  - Inputs and outputs must be of signal data type.

```
ENTITY half_adder
  GENERIC(delays : TIME := 10 ns);
  PORT(x, y : IN BIT;
       carry, result : OUT BIT);
END half_adder;
```

**Architecture**

- Consist of two parts:
  - Declaration part
    - e.g. declaration for data type, signal, component
  - Statement part – statements for organization and/or functional operation of the hardware
    - e.g. signal assignment statements, process statements, component instantiation statements

```
ARCHITECTURE arch_name OF entity_name IS
  -- declarations;
  BEGIN
    -- statements;
  END arch_name;
```

**Architecture – Behavioral Model**

- Explicitly describes how outputs of hardware are calculated with given inputs.

```
ARCHITECTURE half_adder_d OF half_adder IS
  BEGIN
    carry <= x AND y;
    result <= x XOR y after 1 ns;
  END half_adder_d;
```
Architecture – Structural Model

- Implicitly describes how outputs of the hardware are determined through a connection of components.
  - The components have previously been built

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
  port (In1, In2, c_in: in std_logic; sum, c_out: out std_logic);
end full_adder;
architecture struct_FA of full_adder is
  component half_adder port
    (a, b: in std_logic; sum, carry: out std_logic);
  end component;
  component or_2 port
    (a, b: in std_logic; c: out std_logic);
  end component;
signal s1, s2, s3: std_logic;
begin
  H1: half_adder port map (a =>In1, b=>In2, sum=>s1, carry=>s3);
  H2: half_adder port map (a=>s1, b=>c_in, sum=>sum, carry=>s2);
  O1: or_2 port map (a=>s2, b=>s3, c=>c_out);
end struct_FA;
```

Processes

- Can be described in
  - Signal assignment statements
    - E.g. \( \text{Carry} \leftarrow x \text{ and } y \)
  - Process statements
    - Syntax (1)

```
[ process_label : ] PROCESS (sensitivity_list )
  -- process_declarations
BEGIN
  --process_sequential_statements
END PROCESS [ process_label ] :
```

Example -- \( \sum_{i=1}^{x} i^2 \)

```
entity square_sum is
  port(x: in Integer; result: out Integer);
end square_sum;
architecture behav of square_sum is
begin
  square_proc : process (x)
  variable temp: integer;
  begin
    temp := 0;
    for i in 1 to x loop
      temp := temp + i*i;
    end loop;
    result <= temp;
  end process square_proc;
end behav;
```

Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
  port (In1, In2, c_in: in std_logic; sum, c_out: out std_logic);
end full_adder;
architecture struct_FA of full_adder is
  component half_adder port
    (a, b: in std_logic; sum, carry: out std_logic);
  end component;
  component or_2 port
    (a, b: in std_logic; c: out std_logic);
  end component;
signal s1, s2, s3: std_logic;
begin
  H1: half_adder port map (a =>In1, b=>In2, sum=>s1, carry=>s3);
  H2: half_adder port map (a=>s1, b=>c_in, sum=>sum, carry=>s2);
  O1: or_2 port map (a=>s2, b=>s3, c=>c_out);
end struct_FA;
```
Processes (cont.)

- Process statements
  - Syntax (2)

```vhdl
[ process_label : ] PROCESS
  -- process_declarations
BEGIN
  --process_sequential_statements
  -- wait statements
END PROCESS [ process_label ];
```

Example

```vhdl
entity my_testbench is
end my_testbench;

architecture behav of my_testbench is
component my_design
  -- port clause;
end component;
signal clk: std_logic := '0';
Begin
  -- other processes;
  input_gen: process
  begin
    clk <= not clk;
    wait for 1 ns;
  end process
  input_gen;
end behav;
```

Simulation

- To test whether the design is functionally correct.
- Three steps involved in the test
  - Generating input
  - Executing the VHDL model
  - Checking the result

Simulation result

\[ \sum_{i=1}^{x} i^2 \]
Single Cycle Processor Model

- How to model a complicated system such as a processor?
- Refer to a simple model written by Lih Wen Koh
  - Available at ~cs3211/public_html/refs/models/single_cycle_core.zip

```
signal sig_next_pc : std_logic_vector(3 downto 0);
signal sig_curr_pc : std_logic_vector(3 downto 0);
signal sig_one_4b               : std_logic_vector(3 downto 0);
signal sig_pc_carry_out : std_logic;
signal sig_insn : std_logic_vector(15 downto 0);
signal sig_sign_extended_offset : std_logic_vector(15 downto 0);
signal sig_reg_dst : std_logic;
signal sig_reg_write : std_logic;
signal sig_alu_src : std_logic;
signal sig_mem_write : std_logic;
signal sig_mem_to_reg : std_logic;
signal sig_write_register : std_logic_vector(3 downto 0);
signal sig_write_data : std_logic_vector(15 downto 0);
signal sig_read_data_a : std_logic_vector(15 downto 0);
signal sig_read_data_b : std_logic_vector(15 downto 0);
signal sig_alu_src_b : std_logic_vector(15 downto 0);
signal sig_alu_result : std_logic_vector(15 downto 0);
signal sig_alu_carry_out : std_logic;
signal sig_data_mem_out : std_logic_vector(15 downto 0);
```
**IM vs RF vs DM**

**Instruction memory**

```vhdl
entity instruction_memory is
  port ( reset           : in  std_logic;
         clk : in  std_logic;
         read_register_a : in  std_logic_vector(3 downto 0);
         read_register_b : in  std_logic_vector(3 downto 0);
         write_enable    : in  std_logic;
         write_register  : in  std_logic_vector(3 downto 0);
         write_data      : in  std_logic_vector(15 downto 0);
  );

begin
  if (reset = '1') then
    -- initial values of the instruction memory:
    -- insn_0 : load  $1, $0, 0   - load data 0($0) into $1
    -- insn_1 : load  ... $3 into 2($0)   - load data ... $3 into 2($0)
    -- insn_5 : store $4, $0, 3   - store data $4 into 3($0)
    -- insn_6 - insn_15 : noop
    -- end of program
  end if;
end instruction_memory;
```

**Data Memory**

```vhdl
entity data_memory is
  port ( reset        : in  std_logic;
         clk : in  std_logic;
         addr_in : in  std_logic_vector(3 downto 0);
         data_out     : out std_logic_vector(15 downto 0) );

begin
  if (reset = '1') then
    -- initial values of the data memory:
    -- reset to zero
    var_data_mem(0)  := X"0005";
    var_data_mem(1)  := X"0008";
    var_data_mem(2)  := X"0000";
    var_data_mem(3)  := X"0000";
    ... var_data_mem(12) := X"0000";
    var_data_mem(13) := X"0000";
    var_data_mem(14) := X"0000";
    var_data_mem(15) := X"0000";
  end if;
end data_memory;
```

**Register file**

```vhdl
entity register_file is
  port ( reset           : in  std_logic;
         clk : in  std_logic;
         read_register_a : in  std_logic_vector(3 downto 0);
         read_register_b : in  std_logic_vector(3 downto 0);
         write_enable    : in  std_logic;
         write_register  : in  std_logic_vector(3 downto 0);
         write_data      : in  std_logic_vector(15 downto 0);
  );

begin
  if (reset = '1') then
    -- initial values of the register:
    -- reset to zeroes
    var_regfile := (others => X"0000");
  end if;
  -- continuous read of the registers at location read_register_a
  -- and read_register_b
  read_data_a <= var_regfile(var_read_addr_a);
  read_data_b <= var_regfile(var_read_addr_b);
  -- the following are probe signals (for simulation purpose)
  sig_regfile <= var_regfile;
end register_file;
```

**TestBench**

```vhdl
library ENTITY single_cycle_core_testbench IS
END single_cycle_core_testbench;
ARCHITECTURE testbench_arch OF single_cycle_core_tes tbench IS
PROCEDURE    -- clock process for clk
BEGIN
  -- initial values of the registers:
  -- reset to zeroes
  var_regfile <= (others => X"0000");
  -- register write on the falling clock edge
  var_regfile(var_write_addr) := write_data;
  -- the following are probe signals (for simulation purpose)
  sig_regfile <= var_regfile;
END PROCESS;
```

**COMP3211/9211 2016S1 wk2_1 P45**

**COMP3211/9211 2016S1 wk2_1 P46**

**COMP3211/9211 2016S1 wk2_1 P47**

**COMP3211/9211 2016S1 wk2_1 P48**