

## Single Cycle Processor Control

Lecturer: Dr. Hui Annie Guo  
 huig@cse.unsw.edu.au  
 K17-501F (ext. 57136)

## Overview

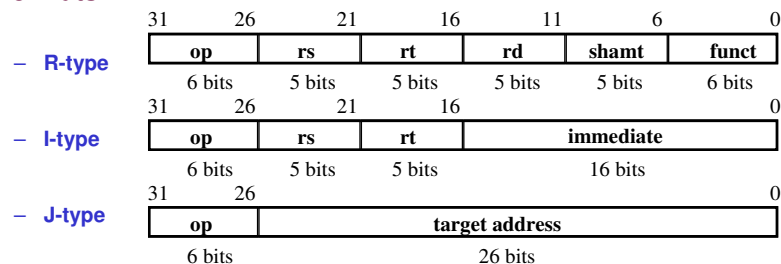
- **Instruction encoding**
- **Control unit design**
  - For single cycle datapath
- **Modeling design with VHDL**
  - Brief Review on VHDL
  - Single Cycle Processor HW Description

COMP3211/9211

2011S1 wk2\_1 P2

## Recall: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:



- The different fields are:
  - 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
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

## Instruction Encoding

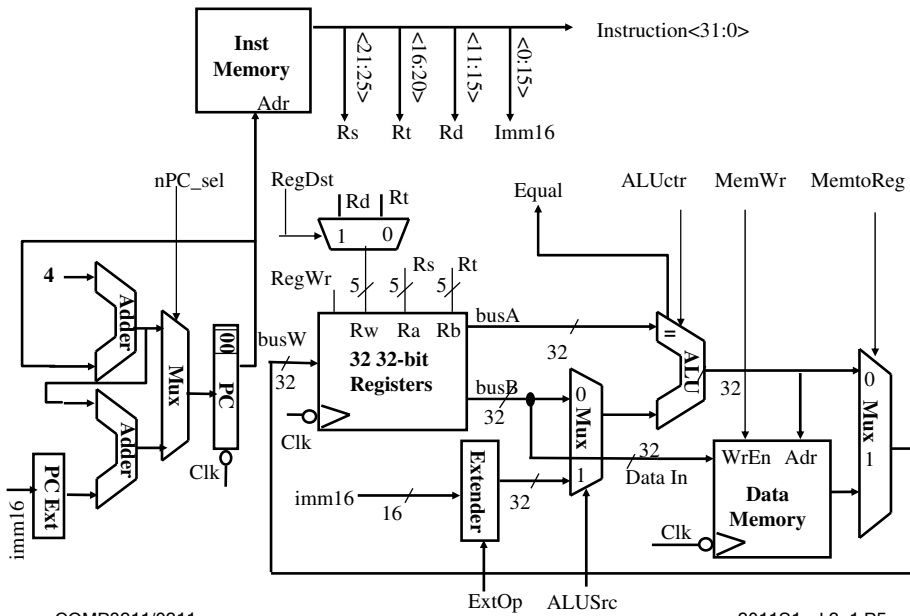
- Instruction encoding is a method that uses binary codes to represent operations and operands.
  - See MIPS reference data sheet in the textbook for MIPS instruction encoding
  - Example

Instr.	R-type	ori	lw	sw	beq	jump
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010

R-type Instr.	add	sub	and	or
func	10 0000	10 0010	10 0100	10 0101

- Control unit design is closely related to instruction encoding

## Control signals



## Step 4: Determine Control Signals

### *inst* Register Transfer

- ADD**  $R[rd] \leftarrow R[rs] + R[rt]; \quad PC \leftarrow PC + 4$   
**ALUSrc** = RegB, **ALUctr** = "add", **RegDst** = rd, **RegWr**, **nPC\_sel** = "+4"
- SUB**  $R[rd] \leftarrow R[rs] - R[rt]; \quad PC \leftarrow PC + 4$   
**ALUSrc** = RegB, **ALUctr** = "sub", **RegDst** = rd, **RegWr**, **nPC\_sel** = "+4"
- ORi**  $R[rt] \leftarrow R[rs] + \text{zero\_ext}(Imm16); \quad PC \leftarrow PC + 4$   
**ALUSrc** = Im, **ExtOp** = "Z", **ALUctr** = "or", **RegDst** = rt, **RegWr**, **nPC\_sel** = "+4"
- LOAD**  $R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign\_ext}(Imm16)]; \quad PC \leftarrow PC + 4$   
**ALUSrc** = Im, **ExtOp** = "Sn", **ALUctr** = "add", **MemtoReg**, **RegDst** = rt, **RegWr**, **nPC\_sel** = "+4"
- STORE**  $\text{MEM}[R[rs] + \text{sign\_ext}(Imm16)] \leftarrow R[rs]; \quad PC \leftarrow PC + 4$   
**ALUSrc** = Im, **ExtOp** = "Sn", **ALUctr** = "add", **MemWr**, **nPC\_sel** = "+4"
- BEQ** if (  $R[rs] == R[rt]$  ) then  $PC \leftarrow PC + \text{sign\_ext}(Imm16)$  || 00 else  $PC \leftarrow PC + 4$   
**nPC\_sel** = EQUAL, **ALUSrc** = RegB, **ALUctr** = "sub"

COMP3211/9211

2011S1 wk2\_1 P6

## Logic for each control signal

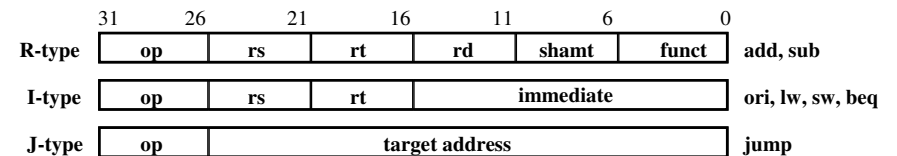
- nPC\_sel**  $\Leftarrow$  if (OP == BEQ) then EQUAL else 0
- ALUSrc**  $\Leftarrow$  if ((OP == "000000") || (OP == BEQ)) then "regB" else "immed"
- ALUctr**  $\Leftarrow$  if (OP == "000000") then funct  
 elseif (OP == ORi) then "OR"  
 elseif (OP == BEQ) then "sub"  
 else "add"
- ExtOp**  $\Leftarrow$  if (OP == ORi) then "zero" else "sign"
- MemWr**  $\Leftarrow$  if (OP == Store)
- MemtoReg**  $\Leftarrow$  if (OP == Load)
- RegWr**  $\Leftarrow$  if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst**  $\Leftarrow$  if ((OP == Load) || (OP == ORi)) then 0 else 1

COMP3211/9211

2011S1 wk2\_1 P7

## A Summary of the Control Signals

Predefined Codes	func						
	10 0000	10 0010	We Don't Care :-)				
	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
<b>RegDst</b>	1	1	0	0	x	x	x
<b>ALUSrc</b>	0	0	1	1	1	0	x
<b>MemtoReg</b>	0	0	0	1	x	x	x
<b>RegWrite</b>	1	1	1	1	0	0	0
<b>MemWrite</b>	0	0	0	0	1	0	0
<b>nPCsel</b>	0	0	0	0	0	1	0
<b>Jump</b>	0	0	0	0	0	0	1
<b>ExtOp</b>	x	x	0	1	1	x	x
<b>ALUctr&lt;2:0&gt;</b>	Add	Subtract	Or	Add	Add	Subtract	xxx

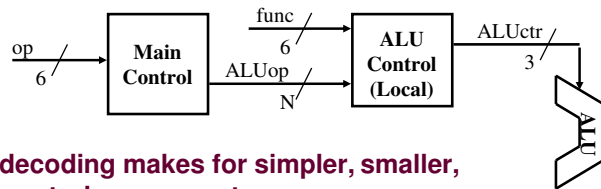


COMP3211/9211

2011S1 wk2\_1 P8

## Local Decoding of ALU control signal

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx

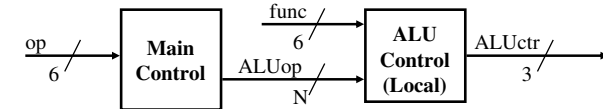


- Local decoding makes for simpler, smaller, faster control components

COMP3211/9211

2011S1 wk2\_1 P9

## The Encoding of ALUop



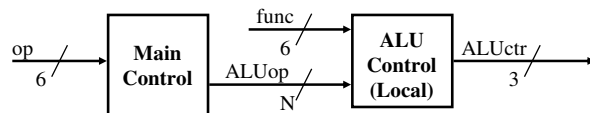
- For the MIPS subset we consider here, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions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

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

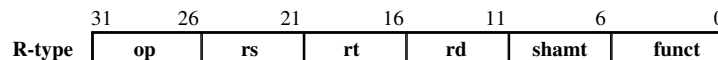
COMP3211/9211

2011S1 wk2\_1 P10

## The Decoding of the "func" Field

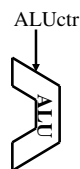


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



ALU control as defined in Ch 4

func<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than



ALUctr<2:0>	ALU Operation
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

COMP3211/9211

2011S1 wk2\_1 P11

## The Truth Table for ALUctr

ALUop (Symbolic)	R-type	ori	lw	sw	beq	func<3:0>	Instruction Op.
	"R-type"	Or	Add	Add	Subtract	0000	add
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	0010	subtract
						0100	and
						0101	or
						1010	set-on-less-than

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

COMP3211/9211

2011S1 wk2\_1 P12

## The Logic Equation for ALUctr<2>

ALUop			func				ALUctr<2>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

$$\text{ALUctr<2>} = \text{!ALUop<2>} \& \text{ALUop<0>} + \\ \text{ALUop<2>} \& \text{!func<2>} \& \text{func<1>} \& \text{!func<0>}$$

## The Logic Equation for ALUctr<1>

ALUop			func				ALUctr<1>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

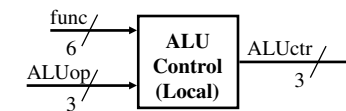
$$\text{ALUctr<1>} = \text{!ALUop<2>} \& \text{!ALUop<1>} + \\ \text{ALUop<2>} \& \text{!func<2>} \& \text{!func<0>}$$

## The Logic Equation for ALUctr<0>

ALUop			func				ALUctr<0>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

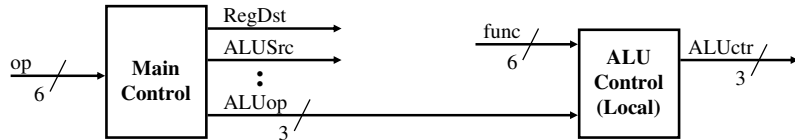
$$\text{ALUctr<0>} = \text{!ALUop<2>} \& \text{ALUop<1>} + \\ + \text{ALUop<2>} \& \text{!func<3>} \& \text{func<2>} \& \text{!func<1>} \& \text{func<0>} + \\ + \text{ALUop<2>} \& \text{func<3>} \& \text{!func<2>} \& \text{func<1>} \& \text{!func<0>}$$

## The ALU Control Block



$$\text{ALUctr<2>} = \text{!ALUop<2>} \& \text{ALUop<0>} + \\ \text{ALUop<2>} \& \text{!func<2>} \& \text{func<1>} \& \text{!func<0>} \\ \text{ALUctr<1>} = \text{!ALUop<2>} \& \text{!ALUop<1>} + \\ \text{ALUop<2>} \& \text{!func<2>} \& \text{!func<0>} \\ \text{ALUctr<0>} = \text{!ALUop<2>} \& \text{ALUop<1>} + \\ \text{ALUop<2>} \& \text{!func<3>} \& \text{func<2>} \& \text{!func<1>} \& \text{func<0>} + \\ \text{ALUop<2>} \& \text{func<3>} \& \text{!func<2>} \& \text{func<1>} \& \text{!func<0>}$$

## The "Truth Table" for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUOp (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUOp <2>	1	0	0	0	0	x
ALUOp <1>	0	1	0	0	0	x
ALUOp <0>	0	0	0	0	1	x

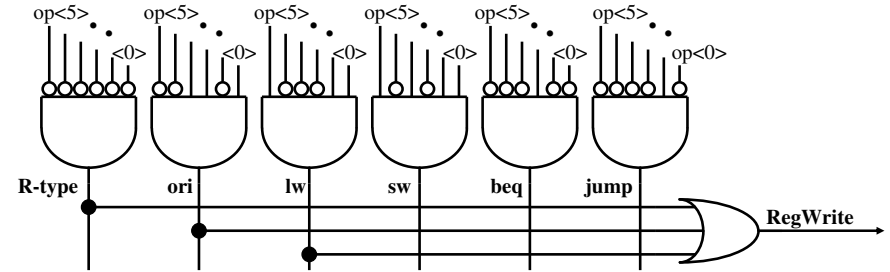
COMP3211/9211

2011S1 wk2\_1 P17

## The "Truth Table" for RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

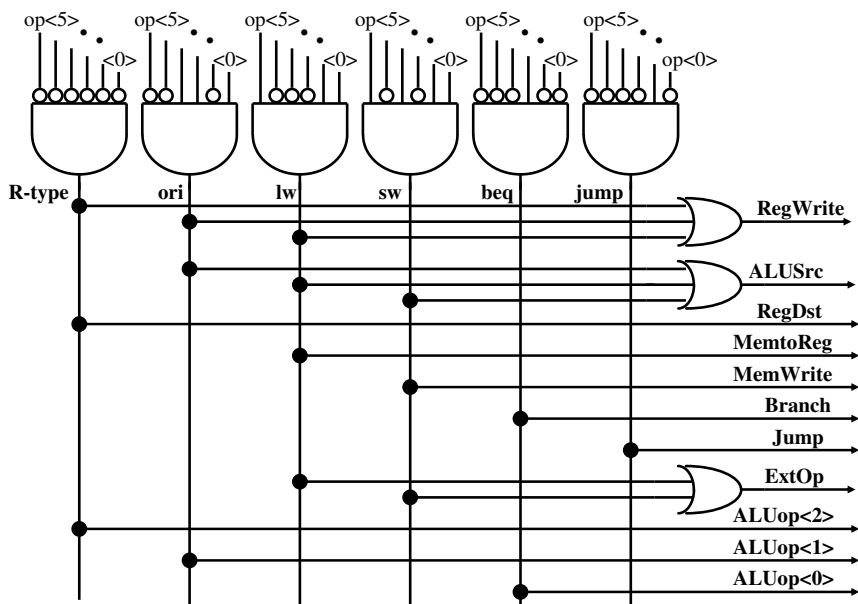
- **RegWrite = R-type + ori + lw**  
 $= \text{op}<5> \& \text{op}<4> \& \text{op}<3> \& \text{op}<2> \& \text{op}<1> \& \text{op}<0> \text{ (R-type)}$   
 $+ \text{op}<5> \& \text{op}<4> \& \text{op}<3> \& \text{op}<2> \& \text{op}<1> \& \text{op}<0> \text{ (ori)}$   
 $+ \text{op}<5> \& \text{op}<4> \& \text{op}<3> \& \text{op}<2> \& \text{op}<1> \& \text{op}<0> \text{ (lw)}$



COMP3211/9211

2011S1 wk2\_1 P18

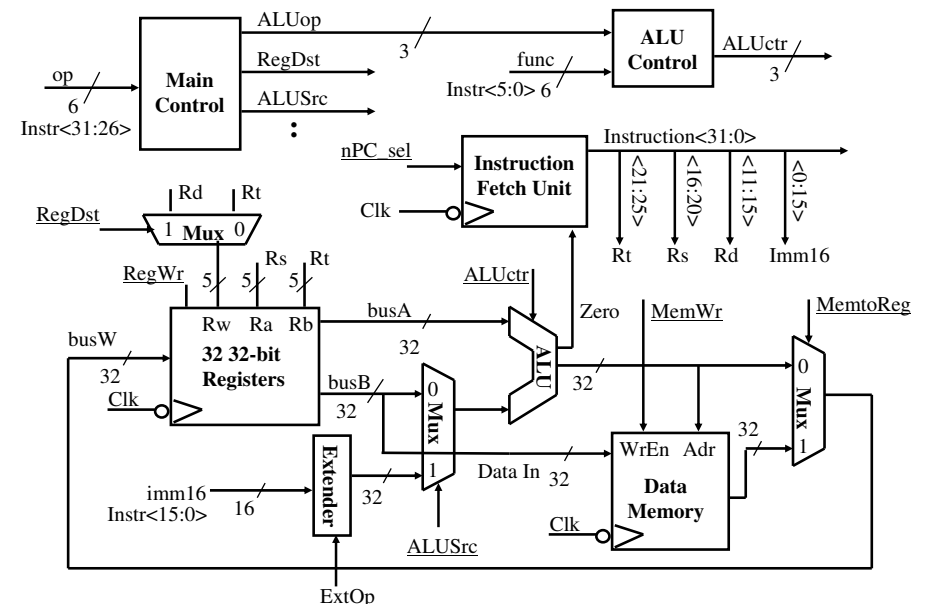
## PLA Implementation of the Main Control



COMP3211/9211

2011S1 wk2\_1 P19

## Putting it All Together: A Single Cycle Processor



COMP3211/9211

2011S1 wk2\_1 P20

## VHDL

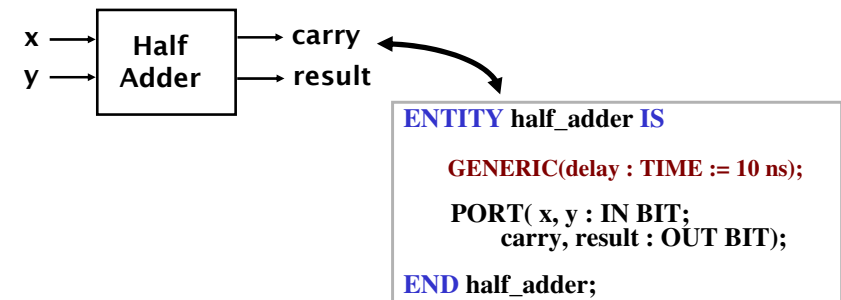
- A Hardware Description Language for Very high speed integrated circuits
- Hardware systems have
  - special attributes
    - Delay and concurrency
  - Interactive relationship between components.
- Basic structure of VHDL code
  - Entity
    - For interface description
  - Architecture
    - For function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

COMP3211/9211

2011S1 wk2\_1 P21

## Entity

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



COMP3211/9211

2011S1 wk2\_1 P22

## 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 ;
```

- Two basic types of descriptions
  - Behavioral model
  - Structural model

COMP3211/9211

2011S1 wk2\_1 P23

## Architecture – Behavioral Model

- Explicitly describes how outputs of the 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;

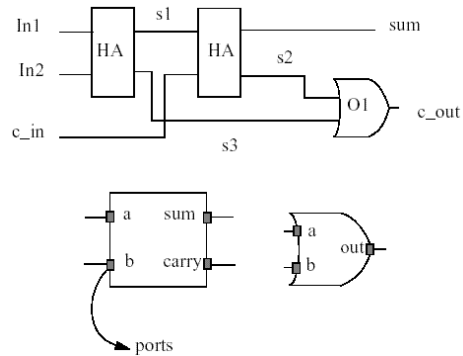
END half_adder_d;
```

COMP3211/9211

2011S1 wk2\_1 P24

## Architecture – Structural Model

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



COMP3211/9211

2011S1 wk2\_1 P25

## Architecture – Structural Model

- 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;
    
```

COMP3211/9211

2011S1 wk2\_1 P26

## Processes

- Can be described in
  - Signal Assignment statements
    - E.g
      - Carry <= x and y
  - Process statements
    - Syntax (1)

```

[ process_label : ] PROCESS (sensitivity_list )

    -- process_declarations

BEGIN

    --process_sequential_statements

END PROCESS [ process_label ] ;
    
```

COMP3211/9211

2011S1 wk2\_1 P27

## 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;
    
```

COMP3211/9211

2011S1 wk2\_1 P28

## Processes (cont.)

- Process statements
  - Syntax (2)

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

COMP3211/9211

2011S1 wk2\_1 P29

## Example

```
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;
```

COMP3211/9211

2011S1 wk2\_1 P30

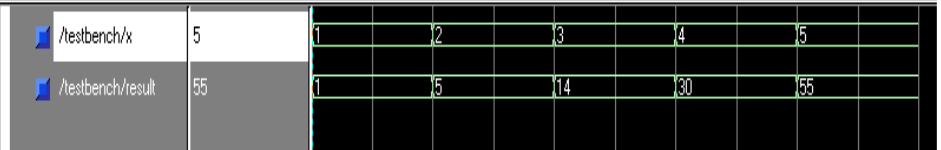
## Simulation

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

COMP3211/9211

2011S1 wk2\_1 P31

$\sum_{i=1}^x i^2$  Simulation result



/testbench/x	5	1	2	3	4	5			
/testbench/result	55	1	5	14	30	55			

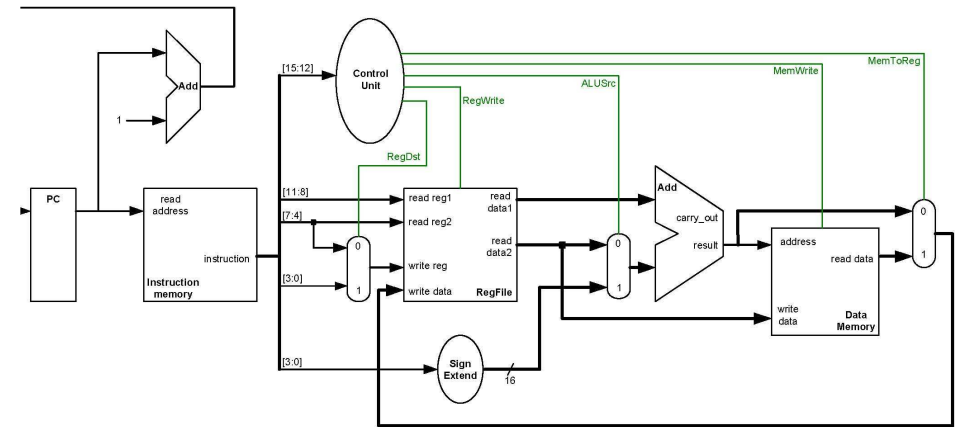
COMP3211/9211

2011S1 wk2\_1 P32

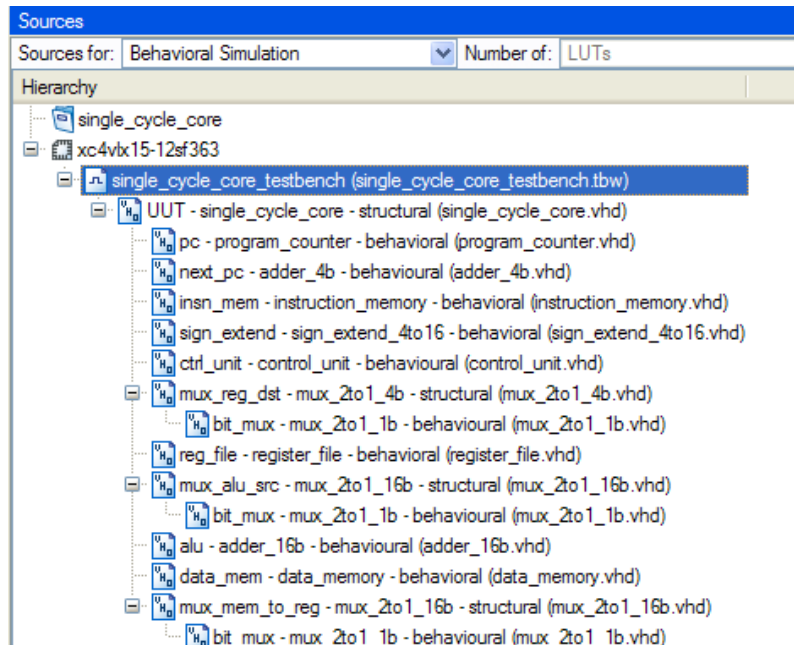
# Single Cycle Processor Model

- How to model a complicated system such a processor?
- Refer to a simple model written by Lih Wen Koh
  - Available at  
[~cs3211/public\\_html/refs/models/single\\_cycle\\_core.zip](http://cs3211/public_html/refs/models/single_cycle_core.zip)

**BLOCK DIAGRAM OF A SINGLE-CYCLE PROCESSOR CORE**



Copyright (C) 2006 by Lih Wen Koh (lwkoh@cse.unsw.edu.au) and Seng Lin Shee (senglin@cse.unsw.edu.au). All Rights Reserved.



```

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

```
entity instruction_memory is
port ( reset : in std_logic;
      clk : in std_logic;
      addr_in : in std_logic_vector(3 downto 0);
      insn_out : out std_logic_vector(15 downto 0) );
end instruction_memory;

architecture behavioral of instruction_memory is

type mem_array is array(0 to 15) of std_logic_vector(15 downto 0);
signal sig_insn_mem : mem_array;

begin
  mem_process: process ( clk,
                       addr_in ) is

    variable var_insn_mem : mem_array;
    variable var_addr : integer;
```

```
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 $2, $0, 1 - load data 1($0) into $2
    -- insn_2 : add $3, $0, $1 - $3 <- $0 + $1
    -- insn_3 : add $4, $1, $2 - $4 <- $1 + $2
    -- insn_4 : store $3, $0, 2 - store 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

  elsif (rising_edge(clk)) then
    -- read instructions on the rising clock edge
    var_addr := conv_integer(addr_in);
    insn_out <= var_insn_mem(var_addr);
  end if;

  -- the following are probe signals (for simulation purpose)
  sig_insn_mem <= var_insn_mem;

end process;

end behavioral;
```

COMP3211/9211

2011S1 wk2\_1 P37

## IM vs RF vs DM

### • Register file

```
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);
      read_data_a : out std_logic_vector(15 downto 0);
      read_data_b : out std_logic_vector(15 downto 0) );
end register_file;

architecture behavioral of register_file is

type reg_file is array(0 to 15) of std_logic_vector(15 downto 0);
signal sig_regfile : reg_file;

begin

  mem_process : process ( reset,
                        clk,
                        read_register_a,
                        read_register_b,
                        write_enable,
                        write_register,
                        write_data ) is

    variable var_regfile : reg_file;
    variable var_read_addr_a : integer;
    variable var_read_addr_b : integer;
    variable var_write_addr : integer;
```

```
begin

  var_read_addr_a := conv_integer(read_register_a);
  var_read_addr_b := conv_integer(read_register_b);
  var_write_addr := conv_integer(write_register);

  if (reset = '1') then
    -- initial values of the registers - reset to zeroes
    var_regfile := (others => X"0000");

  elsif (falling_edge(clk) and write_enable = '1') then
    -- register write on the falling clock edge
    var_regfile(var_write_addr) := write_data;
  end if;

  -- enforces value zero for register $0
  var_regfile(0) := X"0000";

  -- 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 process;

end behavioral;
```

2011S1 wk2\_1 P38

## IM vs RF vs DM

### • Data Memory

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

architecture behavioral of data_memory is

type mem_array is array(0 to 15) of std_logic_vector(15 downto 0);
signal sig_data_mem : mem_array;

begin
  mem_process: process ( clk,
                       write_enable,
                       write_data,
                       addr_in ) is

    variable var_data_mem : mem_array;
    variable var_addr : integer;

  begin
    var_addr := conv_integer(addr_in);
```

```
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(4) := X"0000";
  var_data_mem(5) := X"0000";
  var_data_mem(6) := X"0000";
  var_data_mem(7) := X"0000";
  var_data_mem(8) := X"0000";
  var_data_mem(9) := X"0000";
  var_data_mem(10) := X"0000";
  var_data_mem(11) := 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";

  elsif (falling_edge(clk) and write_enable = '1') then
    -- memory writes on the falling clock edge
    var_data_mem(var_addr) := write_data;
  end if;

  -- continuous read of the memory location given by var_addr
  data_out <= var_data_mem(var_addr);

  -- the following are probe signals (for simulation purpose)
  sig_data_mem <= var_data_mem;

end process;

end behavioral;
```

COMP3211/9211

2011S1 wk2\_1 P39

## TestBench

```
-- libraries

ENTITY single_cycle_core_testbench IS
END single_cycle_core_testbench;

ARCHITECTURE testbench_arch OF single_cycle_core_testbench IS

  COMPONENT single_cycle_core
  PORT (
    reset : In std_logic;
    clk : In std_logic;
  );
END COMPONENT;

SIGNAL reset : std_logic := '1';
SIGNAL clk : std_logic := '0';

SHARED VARIABLE TX_ERROR : INTEGER := 0;
SHARED VARIABLE TX_OUT : LINE;

constant PERIOD : time := 200 ns;
constant DUTY_CYCLE : real := 0.5;
constant OFFSET : time := 0 ns;

BEGIN

  UUT : single_cycle_core
  PORT MAP (
    reset => reset,
    clk => clk
  );
```

```
PROCESS -- clock process for clk
BEGIN
  WAIT for OFFSET;
  CLOCK_LOOP : LOOP
    clk <= '0';
    WAIT FOR (PERIOD - (PERIOD * DUTY_CYCLE));
    clk <= '1';
    WAIT FOR (PERIOD * DUTY_CYCLE);
  END LOOP CLOCK_LOOP;
END PROCESS;

PROCESS
BEGIN
  ----- Current Time: 285ns
  WAIT FOR 285 ns;
  reset <= '0';
  -----
  WAIT FOR 402515 ns;

  IF (TX_ERROR = 0) THEN
    STD.TEXTIO.write(TX_OUT, string("No errors or warnings"));
    ASSERT (FALSE) REPORT
      "Simulation successful (not a failure). No problems detected."
      SEVERITY FAILURE;
  ELSE
    STD.TEXTIO.write(TX_OUT, TX_ERROR);
    STD.TEXTIO.write(TX_OUT,
      string(" errors found in simulation"));
    ASSERT (FALSE) REPORT "Errors found during simulation"
      SEVERITY FAILURE;
  END IF;
END PROCESS;

END testbench_arch;
```

COMP3211/9211

2011S1 wk2\_1 P40