ELEC6234
Embedded Processor Design

FPGA synthesis of a microprocessor core
Synthesis of registers and memory
picoMIPS block diagram - version 1

pMIPS version 1 – no RAM, no branches

PC control

PC

Program Memory \([p \times n+16]\)

Decoder

\([n+15:n+10]\)

ALU flags

\([n+10:n+5]\)

\([n+4:n]\)

\([n+16]\)

\([n+10:n+5]\)

\([n+16]\)

\([n+16]\)

\([n+4:n]\)

\([n+15:n+10]\)

\([n+10:n+5]\)

\([n+4:n]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)

\([n+16]\)
Sequential logic synthesis

- Registers and memory in processors
  - Program Counter
  - General Purpose Register File
  - Instruction Register (not in picoMIPS)
  - RAM – sequential RAM in modern FPGAs

- when writing code for sequential logic, use always_ff blocks with incomplete conditional statements;
  - use always_latch if you want transparent latches; not recommended in modern FPGAs

- avoid temptation to mix complex combinational and sequential logic in one block; partition your design and use multiple always blocks

- for state machines use edge-triggered flip-flops (always_ff)
Edge-triggered flip-flops

// rising edge triggered D-type flip-flop with no reset
module ff1 (input logic clk, d, output logic q);

always_ff @ (posedge clk)
q <= d;
endmodule

// falling edge triggered D-type flip-flop with active-high asynchronous clear
module ff2 (input logic clk, D, Clear, output logic Q);

always_ff @ (negedge clk or posedge Clear)
if (Clear)
Q <= 1'b0;
else Q <= D;
endmodule
// rising edge triggered D-type flip-flop with active-high synchronous preset
module ff3 (input logic clk, D, Preset, output logic Q);

always_ff @ (posedge clk)
    if (Preset)
        Q <= 1'b1;
    else
        Q <= D;
endmodule

// rising edge triggered D-type flip-flop with clock enable
module ff4 (input logic clk, D, CE, output logic Q);

always_ff @ (posedge clk)
    if (CE)
        Q <= D;
endmodule
RAM synthesis

- RAMs used in FPGA designs are typically arrays of latches with a separate input and a separate output data bus, an address bus and a Write Enable signal; read is asynchronous.
- Modern FPGAs (e.g. Cyclone V) support only synchronous memory, where both read and write is synchronous.

- Note that there many types of RAMs, eg.
  - RAM with synchronous read
  - RAM with one Enable controlling both ports
  - RAM with separate Enables controlling each port
  - Multiple-Port RAMs

- Synthesis tools are usually able to infer the correct type of RAM supported by the target FPGA, regardless of the SystemVerilog description, but it is useful to be familiar with the specific types of RAM supported by the target FPGA.
// 128-byte RAM
module ram128x8 (input logic we, input logic [6:0] address,
               input logic [7:0] din, output logic [7:0] dout);

logic [7:0] ram [127:0]; // this 2-dimensional array defines the ram memory

// write block
always_latch
  if (we)
    ram[address] <= din;

// asynchronous read block
assign dout = ram[address];

endmodule

Quartus 2 synthesis summary
Family  Cyclone V
Logic utilization  N/A
Combinational ALUTs  1,512
Memory ALUTs  0
Dedicated logic registers  0
Total registers  0
Total pins  24
Total virtual pins  0
Total block memory bits  0

Quartus 2 synthesis summary
Family  Cyclone IV GX
Total logic elements  1,854
Total combinational functions  1,854
Dedicated logic registers  0
Total registers  0
Total pins  24
Total virtual pins  0
Total memory bits  0

Note: no memory bits have been used in the synthesis!
Synchronous RAM blocks in modern FPGAs

Altera Cyclone IV devices feature memory structures that consist of M9K memory blocks that can be configured to provide RAM, shift registers, ROM, and FIFO buffers.

An M9K block contains 8,192 memory bits and is synchronous, i.e. requires a clock.

Altera Cyclone V devices contain two types of memory blocks. Both are synchronous.

1. M10K blocks—10-kilobit (Kb) blocks for larger memory configurations.

2. Memory logic array blocks (MLABs)—640-bit memory blocks for small memories. Each MLAB can be configured as ten 32 x 2 blocks, giving one 32 x 20 simple dual-port SRAM.
// synchronous RAM, 128x8
module ram128x8sync(
    output logic [7:0] dout,
    input logic [6:0] address,
    input logic [7:0] din,
    input logic we, clk);
logic [7:0] mem [127:0];
always_ff @(posedge clk)
begin
    if (we)
        mem[address] <= din; // memory write
    dout <= mem[address]; // synchronous memory read
end
endmodule

Quartus 2 synthesis summary
Family       Cyclone V
Logic utilization    N/A
Combinational ALUTs  0
Memory ALUTs        0
Dedicated logic registers  0
Total registers    0
Total pins         25
Total virtual pins 0
Total block memory bits  1,024

Quartus 2 synthesis summary
Family       Cyclone IV GX
Total logic elements       0
Total combinational functions  0
Dedicated logic registers  0
Total registers    0
Total pins         25
Total virtual pins 0
Total memory bits  1,024
General Purpose Registers

For details see file regs.sv

From instruction code

From ALU or RAM

CLK

Write is synchronous

To ALU
Altera Quartus synthesis of registers

Top-level Entity Name: regs
Family: Cyclone IV GX
Total logic elements: 386 / 14,400 (3 %)
Total combinational functions: 386 / 14,400 (3 %)
Dedicated logic registers: 256 / 14,400 (2 %)
Total registers: 256
module pc #(parameter Psize = 6) // up to 64 instructions
(input logic clk, reset, PCincr, PCabsbranch, PCrelbranch,
input logic [Psize-1:0] Branchaddr,
output logic [Psize-1:0] PCout);
//------------- code starts here---------
logic[Psize-1:0] Rbranch; // temp variable for addition operand
always_comb // multiplexer to select next instruction offset
if (PCincr) // see always_ff block below
    Rbranch = { {(Psize-1){1'b0}}, 1'b1}; // add 1
else Rbranch = Branchaddr; // add branch addr

always_ff @ (posedge clk or posedge reset) // async reset
if (reset) // reset
    PCout <= {Psize{1'b0}};
else if (PCincr | PCrelbranch) // increment or branch relative
    PCout <= PCout + Rbranch; // 1 adder handles both
else if (PCabsbranch) // absolute branch, load branch addr
    PCout <= Branchaddr;
endmodule // module pc
Altera Quartus Synthesis of Program Counter

Top-level Entity Name: pc
Family: Cyclone IV GX
Total logic elements: 14
Total combinational functions: 14
Dedicated logic registers: 6
Total registers: 6
Program Memory

module prog #(parameter psize = 6, isize = 24) // psize - address width, isize - instruction width
  (input logic [psize-1:0] address,
   output logic [isize:0] instr); // l - instruction code

// program memory declaration, note: 1<<n is same as 2^n
logic [isize:0] progMem[ (1<<Psize)-1:0];

// get memory contents from file
initial
  $readmemh("prog.hex", progMem);

// program memory read
  assign instr = progMem[address];

endmodule // end of module prog

Notes:
1) program file name must be prog.hex
2) use 1<<n to calculate 2^n
Sample prog.hex

// sample picoMIPS program 2
// n = 8 bits, Lsize = 16+n = 24 bits
// format: 6b opcode, 5b %d, 5b %s, 8b immediate or address

// HEX ///////////////////////////// BINARY ////////////////////////////////////////////////////////////////////// ASSEMBLER ///////////
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
282005 // 24'b0010_1000_0010_0000_0000_0101 ADDI %1, %0, 5; load 5 in register 1
284007 // 24'b0010_1000_0100_0000_0000_0111 ADDI %2, %0, 7; load 7 in register 2
082200 // 24'b0000_1000_0010_0010_0000_0000 ADD %1, %2; %1 = %1 + %2
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP

Notes:
1) syntax is simple: each line contains instruction code in hex followed by a comment

2) this is a simple test that uses NOP, LDI (Load Immediate to Register) and ADD instructions

3) LDI is a ‘synthetic’ instruction; LDI %d, imm is implemented as: ADDI %d, %0, imm
Summary

• Progress so far:
  – Data path
    • ALU, General Purpose Registers,
  – Control path
    • Program Counter, Program Memory,
  – Next step:
    • Develop a simple decoder and a CPU encapsulating module to allow integration of modules developed so far and simple program execution