ENEL353 Computer Hardware Engineering I (including ENEL340) Digital Logic Design Project

Andrew Bainbridge-Smith, A404

August 5, 2010

1 Objective

The purpose of the assignment is to develop a good understanding of digital logic design using VHDL. The emphasis is on both control and computation with a development of state machines with data paths. This is RTL design.

The skills developed will be demonstrated by creating a small microprocessor system on an FPGA development kit, and communicated through a group report.

You will be working in teams of five (5) or six (6).

2 Overview

Each team will be required to develop a rather primitive computer system consisting of at least the following 4 identifiable components:

- A CPU core — execution unit and ALU with defined instruction set.
- A MMU — a memory management unit to handle bus requests for instructions and data.
- An IO control unit — a unit to handle the requests.
- The buses — connecting the units above together.

The work must be implemented using VHDL and demonstrated on a Xilinx Spartan 3-200 FPGA development kit. You may use a schematic for the top level, but this should have minimal components on it.

The team should collectively work on the bus design. The team must then be broken up into smaller units (pairs), with each sub-group working a different module for the remaining units.

3 The CPU

The CPU is an 8-bit word-length RISC styled machine. It consists of the following register files:

- 8 general purpose registersb (8-bit): r0...r7
- 3 address registers (16-bit): a0...a2
- Special purposes registers (16-bit): PC (program counter), SR (status register)

The CPU must implement a minimum 24 instruction set, detailed below. The instruction word-length is 16-bit.

Instruction	Operand 1	Operand 2	Description
LDI	r0...r3	8-bit immediate	Load Immediate
LD	r0...r7	a0...a2	Load indirect
	r0...r7	ax+ (x= 0...2)	Load indirect, auto increment
	r0...r7	ax- (x= 0...2)	Load indirect, auto decrement
STI	a0...a2	8-bit immediate	Store immediate
ST	a0...a2	r0...r7	Store indirect
	ax+ (x= 0...2)	r0...r7	Store indirect, auto increment
	ax- (x= 0...2)	r0...r7	Store indirect, auto decrement
MV	r0...r7	r0...r7	Move
	ax(HL) (x= 0...2)	r0...r7	Move
	r0...r7	ax(HL) (x= 0...2)	Move
AND	r0...r7	r0...r7	Bitwise and
OR	r0...r7	r0...r7	Bitwise or
NOT	r0...r7	r0...r7	Bitwise not
XOR	r0...r7	r0...r7	Bitwise xor
ADD	r0...r7	r0...r7	Addition
ADC	r0...r7	r0...r7	Addition with carry
SUB	r0...r7	r0...r7	Subtraction
SBB	r0...r7	r0...r7	Subtraction with borrow (carry)
NEG	r0...r7	r0...r7	Negation (2's complement)
CMP	r0...r7	r0...r7	Compare
BEQ	8-bit immediate		branch if equal
BNE	8-bit immediate		branch if not equal
BLT	8-bit immediate		branch if less than
BGT	8-bit immediate		branch if greater than
BC	8-bit immediate		branch if carry
BNC	8-bit immediate		branch if no carry
RJMP	8-bit immediate		relative unconditional branch
JMP	a0...a2		unconditional branch
NOP			No instruction

The RTL and instruction codes are given in Appendix A. If there is time and space you may embellish the CPU design.

The CPU is based on a Harvard principle — separate program and data stores — and memory mapped IO. Little endian is to be used. With exception of the load and store instructions, each instruction takes the same fixed number of clock cycle to complete (this is 1 if the design is pipelined).

4 The MMU

The memory management unit is responsible for handling memory access requests from the CPU. As a Harvard design is used there will be two separate busses. All accesses are to be queued and channelled over an RS232 link to a host PC. The host PC will be responsible for actual fetches (and storing) of memory for both instructions and data.

The memory space for program instruction is 12-bits in size. The memory space for data (excluding IO) is 15-bits in size. The least significant bit in the 16-bit data address is used to indicate if the data access is to data memory or IO. IO addresses have a zero, while memory addresses have a 1.

Data packets on the RS232 link should follow this protocol:

- 8-bit header:
  - bit 7: read=1, write=0
  - bit 6-4: reserved
  - bit 3-2: diagnostic modes
  - bit 1: fetch-return=1, otherwise 0
  - bit 0: instruction bus=1, data bus=0
- 16-bit address
- 8/16-bit data (optional):
  The size of the packet is determined by: if it is a data bus request (8-bit) or an instruction bus request (16-bit). This component of the packet will only be present if a write is being performed, or fetch-return is high and the PC is responding with a read request.

It is not necessary to write a PC side application to handle the memory requests but this is probably a good idea. This need not be written by the team members working on the MMU module.

You should use a predefined RS232 module, these can be found for example from OpenCores.

While the description of the MMU is simple, in reality it is likely to be as complex as the CPU to implement.

5 IO Unit

A computer without IO is just a heater. In this unit it is necessary to supply some IO to the computer system. This must as a minimum provide one LED and one push button. The push button must be properly switch debounced. The group should think careful about what IO features they would like to provide.

6 Common Buses

The design of the buses is essentially the specification of the entities of the other units.

The instruction bus is:

- 12-bit address
- 16-bit data
- 1-bit fetch request. Pulled low to request when address valid and high to acknowledge data latched.
- 1-bit fetch acknowledge. Pulled low when data ready and high when fetch request pulled low.

The data bus is:

- 16-bit address
- 8-bit data
- 1-bit read (not write)
- 1-bit bus request. Pulled low to request use of bus (address, data and read lines driven at same time) if state of the other lines are the same no bus contention found and continue, otherwise release. Pulled high at the completion of the bus cycle.
- 1-bit bus acknowledge. Pulled low when data read (for write cycle) or low when data ready (for read cycle). Pulled high when bus request is high. The read/write must occur on the second cycle of the bus to ensure no errors arising from bus contention.

Your group may choose to modify the design of the buses. The implementation of the buses should be assigned to even the workload of the team members (probably the IO subgroup).

7 Report

Each team must produce a report that summarises the design work. The report should clearly state not only what was done, but why. There should be a separate section for each component of the work, with appropriate illustrations and reference to the appendices that contain all the program code and simulation test-beds etc. There should also be a section on the integration of the design units into the final solution, plus some commentary on whether it works. A final section should be reserved to comment on what you learned from doing the exercise. I expect to see a full (and lengthy) reference list.

The main body of the report (ie. excluding the appendices) should not exceed 24 pages, and be written in an 11pt times font.

It should be clearly stated who authored which parts of the work, i.e. all program code should have names in it. There should also be a preface to the report, stating who did what (coding and writing) with a percentage estimate of their contribution to the work. Ideally the preface should be a single page and not exceed 2 pages.

The report and code work is worth 60% of the final mark.

8 Demonstration

Each team will need to demonstrate their work to me - appointments to be made later in term 4.

Demonstrations will consist of 30% of the final mark, and will be based on completeness of the design, thoroughness, any embellishments, testing and organisational skills.

9 Individual Contribution

Up to 10% of the mark of the mark will be attributed to individual contribution, this will be assessed by interactions in the demonstration, contribution to the department wiki page, and similar activities.

10 Submissions

You will be required to submit 2 electronic files using “Learn”:

- A pdf of your report.
- A zip file of the xilinx design.

The zip file should contain all the files necessary to synthesize and implement the work on the development kit -i.e. schematics, vhdl and ucf files.

11 Support

There will be a postgrad assigned to be available on Mondays for 1hr each week in the electronics lab to help with technical questions. I also encourage the use of the WIKI made available on “Learn” for this course.

12 Housekeeping

1. Book Spartan 3-200 development kits from Philipp Hof (DSL Technician)
2. The practical demonstration, attended by all members of the group, will be given at some time on Monday 12th October. Demonstrations will last for approximately 20-30 minutes. Booking times will be arranged at a later date.
3. The final report, and zip files are to be submitted by Friday 16th October at 10pm using the "Learn" system.

Appendix A - CPU RTL Description

This appendix outlines each 16-bit instruction in the CPU. The top most 6-bits of the opcode encode the instruction, while the remaining 10-bits encode the operand address and/or immediate value.

Load Immediate

Syntax

LDI rx, v

RTL

rx ← v, x = 0..3

Opcode

1	0	0	0	0	1	v7	v6	v5	v4	v3	v2	v1	v0	x1	x0

Load Direct

Syntax

LD rx, ay

RTL

rx ← [ay], x= 0..7, y-0..2

Opcode

0	0	0	0	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Load Direct - auto increment

Syntax

LD rx, ay+

RTL

rx ← [ay]; ay ← ay+1, x= 0..7, y=0..2

Opcode

0	0	1	0	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Load Direct - auto decrement

Syntax

LD rx, ay-

RTL

rx ← [ay]; ay ← ay-1, x= 0..7, y=0..2

Opcode

0	1	0	0	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Store Immediate

Syntax

STI ay, v

RTL

[ay] ← v, y=0..2

Opcode

1	0	0	1	0	1	v7	v6	v5	v4	v3	v2	v1	v0	y1	y0

Store Indirect

Syntax

ST ay, rx

RTL

[ay] ← rx, x=0..7, y=0..2

Opcode

0	0	0	1	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Store Indirect - auto increment

Syntax

ST ay+, rx

RTL

[ay] ← rx; ay ← ay+1, x=0..7, y=0..2

Opcode

0	0	1	1	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Store Indirect - auto decrement

Syntax

ST ay-, rx

RTL

[ay] ← rx; ay ← ay-1, x=0..7, y=0..2

Opcode

0	1	0	1	0	1	0	0	0	y1	y0	0	0	x2	x1	x0

Move

Syntax

MV ry, rx

RTL

ry ← rx, x,y=0..7

Opcode

0	0	0	1	0	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Move

Syntax

MV ay(HL), rx

RTL

ayn ← rx, x=0..7, y=0..2, n=1 for high byte otherwise low byte

Opcode

0	0	0	1	0	0	1	n	0	y1	y0	0	0	x2	x1	x0

Move

Syntax

MV rx, ay(HL)

RTL

rx ← ayn, x=0..7, y=0..2, n=1 for high byte otherwise low byte

Opcode

0	0	0	1	0	0	0	0	x2	x1	x0	1	n	0	y1	y0

Bitwise And

Syntax

AND ry, rx

RTL

ry ← ry AND rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	0	0	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Bitwise Or

Syntax

OR ry, rx

RTL

ry ← ry OR rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	0	1	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Bitwise NOT

Syntax

OR ry, rx

RTL

ry ← NOT rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	1	0	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Bitwise Exclusive Or

Syntax

XOR ry, rx

RTL

ry ← ry XOR rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	1	1	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Addition

Syntax

ADD ry, rx

RTL

ry ← ry + rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	1	0	0	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Addition with carry

Syntax

ADC ry, rx

RTL

ry ← ry + rx + C; SR ← {Z,C,N}, x,y=0..7

Opcode

0	1	0	1	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Subtraction

Syntax

SUB ry, rx

RTL

ry ← ry - rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	1	1	0	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Subtraction with borrow (carry)

Syntax

SBB ry, rx

RTL

ry ← ry - rx - C; SR ← {Z,C,N}, x,y=0..7

Opcode

0	1	1	1	1	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Negation - 2's complement

Syntax

NEG ry, rx

RTL

ry ← -rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	1	0	0	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Compare

Syntax

CMP ry, rx

RTL

T ← ry - rx; SR ← {Z,C,N}, x,y=0..7

Opcode

0	0	1	1	0	0	0	0	y2	y1	y0	0	0	x2	x1	x0

Branch If Equal

Syntax

BEQ v

RTL

if (Z=1) PC ← PC + v;

Opcode

1	0	0	0	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Branch If Not Equal

Syntax

BNE v

RTL

if (Z=0) PC ← PC + v;

Opcode

1	0	0	1	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Branch If Less Than

Syntax

BLT v

RTL

if (N=1 and Z=0) PC ← PC + v;

Opcode

1	0	1	0	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Branch If Greater Than

Syntax

BGT v

RTL

if (N=0 and Z=0) PC ← PC + v;

Opcode

1	0	1	1	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Branch If Carry

Syntax

BC v

RTL

if (C=1) PC ← PC + v;

Opcode

1	1	0	0	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Branch If NOT Carry

Syntax

BNC v

RTL

if (C=0) PC ← PC + v;

Opcode

1	1	0	1	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Relative Unconditional Jump

Syntax

RJMP v

RTL

PC ← PC + v;

Opcode

1	1	1	0	1	1	v7	v6	v5	v4	v3	v2	v1	v0	0	0

Unconditional Jump

Syntax

JMP ay

RTL

PC ← ay, y=0..2

Opcode

0	1	1	1	1	1	0	0	0	y1	y0	0	0	0	0	0