Simple Multi-Cycle Processor

Overview

This implementation of a simple multi-cycle processor consists of the datapath, control (FSM), and various inputs and outputs. The processor supports only fixed-length 8-bit instructions and data, and can only access memory one byte at a time. Overall, 10 instructions are implemented in hardware.

Functional Description

The inputs and outputs include:

- Inputs:
  - reset – clears the values of all registers and control signals, resets the condition flags, and resets the control FSM (KEY[0])
  - clock – all writes and cycle transitions happen on the positive edge of the clock (KEY[1])
  - HEXsel[1..0] – selects one of the four registers to be displayed on the hex displays; this exists for the users of the DE1 board, which does not have many hex displays. The chooseHEXs circuit component can still be used to display four values simultaneously (for example, of the four registers), but that requires eight hex displays, available only on the DE2. Users of the DE2 can make the necessary circuit connections if they so desire. (SW[17..16])

- Outputs:
  - PCWrite – LEDR[17]
  - AddrSel – LEDR[16]
  - MemRead – LEDR[15]
  - MemWrite – LEDR[14]
  - IRLoad – LEDR[13]
  - R1Sel – LEDR[12]
  - MDRLoad – LEDR[11]
  - R1R2Load – LEDR[10]
  - ALU1 – LEDR[9]
  - ALU2[2..0] – LEDR[8..6]
  - ALUOp[2..0] – LEDR[5..3]
  - ALUOutWrite – LEDR[2]
  - RFWrite – LEDR[1]
  - RegIn – LEDR[0]
  - FlagWrite – LEDG[7]
  - N (negative) – LEDG[1]
  - Z (zero) – LEDG[0]
  - Register Display
    - upper 4 bits HEX1[6..0]
    - lower 4 bits HEX0[6..0]
    - Note: as mentioned previously, the circuit can be modified

to support the remaining 6 hex displays, available on the DE2 board.

Instructions (Hardware)

- Addition (add)
- Subtraction (sub)
- NAND (nand)
- Shifting (shift)
- OR with Immediate (ori)
- Load (load)
- Store (store)
- Branch If Positive Zero (bpz)
- Branch If Zero (bz)
- Branch If Not Zero (bnz)

Assembler An assembler is provided with the processor to aid students with the process of creating simple programs. It generates a Memory Initialization File (MIF) that can be used to initialize the memory of the processor design. The assembler can be invoked in the shell (after logging in using SSH) by typing “asm”. If this does not work, make sure that the assembler is located in the working directory or is configured to run properly. The assembler accepts one mandatory argument and one optional argument. The first argument is the name of the file containing the assembly program, and must be provided. It can have any file extension, but we suggest using “.s”. The second argument is optional, and is the name of the output MIF file. If it is not provided, the default output filename is always “data.mif”. This name was chosen because the Quartus implementation of the processor assumes “data.mif”, located in the same directory as the design, to be the initialization file. This can be changed, however, using a wizard in Quartus.

The assembler is based on the concept of columns, similar to many other assemblers. The first column begins at the first character of a line, the second column is separated from the first by any amount of whitespace, and the third column is separated from the second in the same way. If more columns are present, the assembler will issue an error, unless the text immediately after the third column is a comment.

Comments are indicated using a semicolon. The assembler will ignore anything following a semicolon, unless the semicolon interrupts an instruction or directive, in which case an error will be issued.

The first column must either be blank, or contain a unique label. Please note that this assembler only allows labels on the same line as the instruction/directive to which they refer. A label cannot appear on a line by itself.

The second column must contain a valid keyword identifying the instruction or directive. Finally, the third column must contain the parameters/operands. Please note that in this assembler no spaces are allowed in the third column. If two operands are required, they are separated using a comma without spaces in between.

Numerical constants can be specified in binary, octal, decimal, or hexadecimal format. Binary values are preceded with a '%' sign, octal values are preceded by an initial zero (0), and hexadecimal values are preceded with a '$' sign. If none of these characters precede the number, it is assumed to be in decimal format. Please note that negative signs are only supported when using the decimal format.

Overall, 14 keywords are supported:

- add r1,r2
- sub r1,r2
- nand r1,r2
- shift r1,imm3
  - shiftl r1,imm2
  - shiftr r1,imm2
- ori imm5
- load r1,(r2)
- store r1,(r2)
- bpz ADDR
- bz ADDR
- bnz ADDR
- org imm8
- db imm8

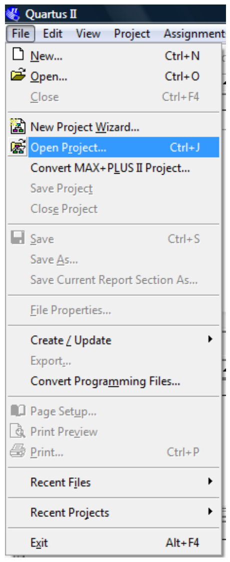
The shift instruction, as described in the course notes, accepts a 3-bit value. Bit #2 determines the direction (0 – right, 1 – left), and bits #1 and #0 determine the number of shift positions. For convenience, "shiftl" and "shiftr" are provided to shift left and right, respectively. They accept a number between 0 and 3, specifying the number of shift positions. They are not true instructions, because they both get converted to an equivalent "shift" instruction. Please note that imm2, imm3, and imm5 must be unsigned positive numbers (i.e. negative signs are not allowed). The brackets around "r2" for "load" and "store" are also required. ADDR can be either a previously defined label, or a 4-bit signed number. Please note that the effective branch address is calculated as follows: current_address + imm4 + 1. Do not forget about the 1 that is added, especially when using backward branching. "org" can be used to set the current address. It is specified here to use imm8, but the actual width of the number depends on the memory size. For this lab, a number between 0 and 255 can be used. "db" is used to place a single byte of at the current address. The byte can be either an

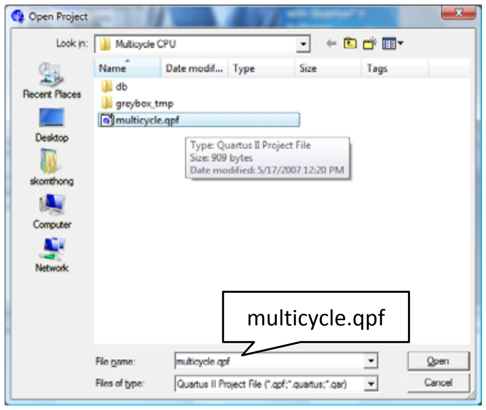
8-bit number, or a character in single quotes. Only one byte can be specified per "db" directive. While "org" and "db" are not instructions, they can be associated with a label, identifying the new address that they define (in the case of "org"), or the address at which they place their data (in the case of "db"). Please note that a label associated with an "org" statement and with an instruction or directive immediately following it identify the same address, unless another "org" statement is the following directive.

Changing Initial Memory Contents

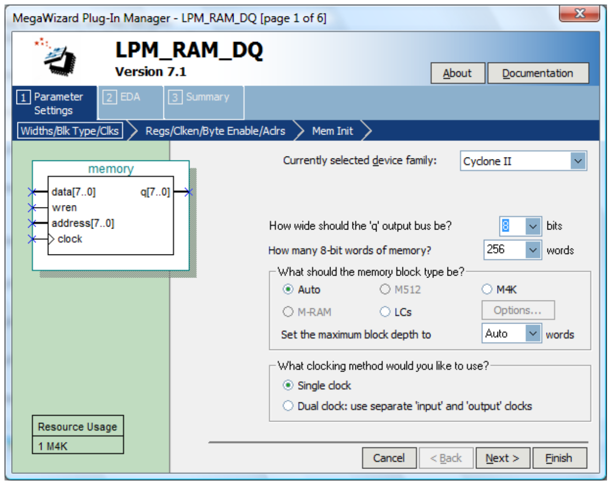
By default, the memory block used in the design assumes that its initial contents come from a MIF file called "data.mif". However, it may sometimes be useful to be able to change this to a different file name.

1. To change the initial contents of the memory, first open the Quartus project called "multicycle.qpf".





2. Double click on the component labeled "Data Memory".



Currently selected device family: Cyclone II

How wide should the 'q' output bus be? 8 bits

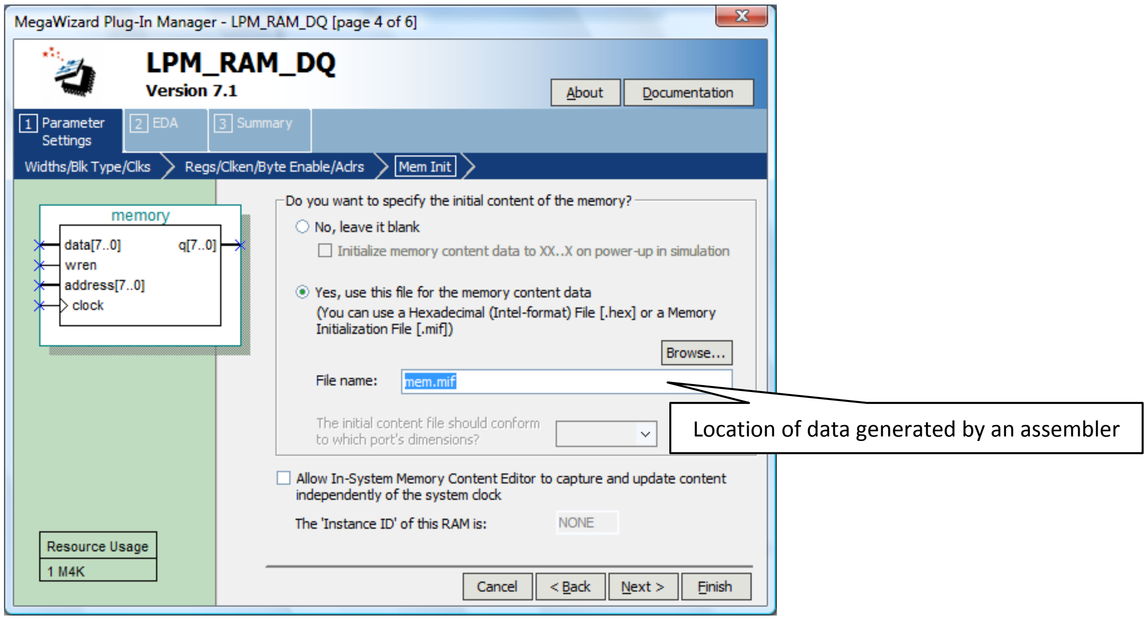
How many 8-bit words of memory? 256 words

What should the memory block type be? [✓ Auto] M512 M4K M-RAM LCs

Set the maximum block depth to Auto words

What clocking method would you like to use? [✓ Single clock] Dual clock: use separate 'input' and 'output' clocks

3. Select “Mem Init”, and locate the MIF file generated by the assembler.


MegaWizard Plug-In Manager - LPM_RAM_DQ [page 4 of 6]

LPM_RAM_DQ
Version 7.1

Do you want to specify the initial content of the memory?

- ○ No, leave it blank
  - ☐ Initialize memory content data to XX..X on power-up in simulation
- ● Yes, use this file for the memory content data
  (You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

File name: mem.mif

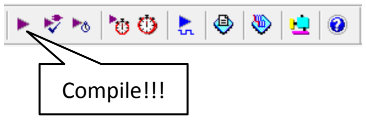
The initial content file should conform to which port's dimensions?

☐ Allow In-System Memory Content Editor to capture and update content independently of the system clock

The 'Instance ID' of this RAM is: NONE

Resource Usage: 1 M4K

4. Compile the project.



5. Run the program!