16 bit MIPS CPU

By Zhu, Kevin and Zhifei, song

CSCB58W19

Welcome to the mips_16 CPU quick start guide. The first thing you need to consider is the version of Quartus you are using. The mips_16 CPU was designed, tested, and run on Quartus prime pro 16.0.

Each of the modules has been individually tested to ensure that there are no bugs in the operation of the components themselves. The data-path has been extensively tested, however if you are to find any errors or bugs, please feel free to fix them yourself.

The goal of this project was to create something that could be built upon by others to new heights. If you are to have any ideas or projects that you wish to complete using this CPU, please do not hesitate to do so. Also, we encourage modifying this project to your own personal needs.

Now into the hardware.

The mips_16 CPU design was based off the original MIPS CPU processor, as a result it follows most of the MIPS conventions such as being single cycle, and having a reduced instruction set. In this case, the instruction set is larger than what it would be due to the hardware capabilities of the instruction size. Normally, an instruction for an ALU operation would have a single opcode, where you could specify the function of the ALU with some extra function selection bits in the instruction, however since we wanted to have the ability to operate on 32 registers, the ALU function selection has been handed over to the opcode and thus the number of instructions are larger than they would normally be.

The instruction are 16 bits long and format follows a simple format:

For documenting instructions, values [ADDRESS] represent a 10 bit address value for ram, [IMMEDIATE] represents an immediate value which may be 5 or 10 bits depending on the size of the instruction(which must be written in decimal in the assembler), and [REGISTER] represents a 5 bit register address denoted as ($0-$31) when writing assembly.

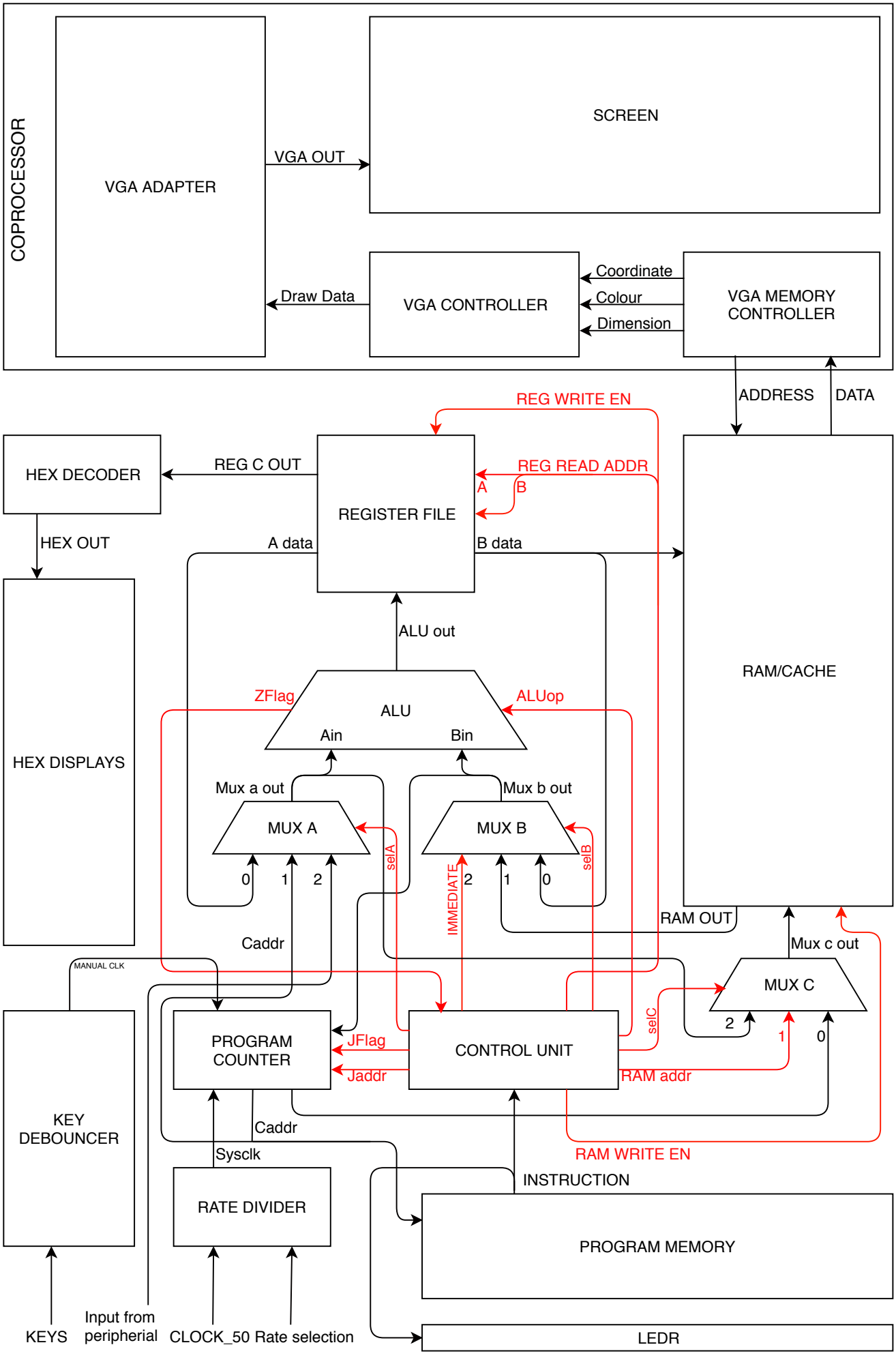
OPCODE (6 bits)	ADDRESS/REGISTERS (10 bits)

And that's all there is to it.

The documentation for the instructions can be seen in the CTRLUNIT.v file. All of the instructions include comments beside them explain their function and purpose. Don't bother with remembering opcodes and addresses though, as the assembler will take care of all that for you.

Keep in mind that the 0^{th} register at address 00000 always contains 0, it is reserved for 0 and cannot be changed. Also remember that register $1 at 00001 always contains the output of any previous arithmetic operation. Since the CPU is 16 bits, there is not enough space in the instruction to address two registers and also a third output register, the solution we decided to implement was to reserve register 00001 so that the output of the ALU will always be saved there, if there was an operation. This does not mean you cannot do ADD $4 $2 $3 however, as the assembler will automatically convert that command into two commands: ADD $2 $3 and MV $1 $4, which adds the values and moves it to the desired register.

Here is an included flowchart diagram of the CPU.


COPROCESSOR
VGA ADAPTER
VGA OUT
SCREEN
Draw Data
VGA CONTROLLER
Coordinate
Colour
Dimension
VGA MEMORY CONTROLLER
ADDRESS
DATA
REG WRITE EN
HEX DECODER
REG C OUT
REGISTER FILE
REG READ ADDR
A
B
HEX OUT
A data
B data
RAM/CACHE
HEX DISPLAYS
ALU out
ZFlag
ALU
ALUop
Ain
Bin
Mux a out
Mux b out
MUX A
MUX B
selA
selB
IMMEDIATE
0
1
2
RAM OUT
Caddr
Mux c out
MANUAL CLK
MUX C
selC
PROGRAM COUNTER
JFlag
Jaddr
CONTROL UNIT
RAM addr
KEY DEBOUNCER
Caddr
Sysclk
RAM WRITE EN
INSTRUCTION
RATE DIVIDER
PROGRAM MEMORY
KEYS
Input from peripherial
CLOCK_50 Rate selection
LEDR

Notice how the input from peripheral wire seems to be coming out of nowhere. This is for you to use. The dedicated instruction SI[REGISTER] allows you to save the value in the wire into a register of your choice in the register file. This will allow for input from the user, or coprocessors, or anything really. Its up to you to decide.

The register file also boasts the ability to dual read out values to MUX A and MUX B asynchronously. You don't need to wait for the clock cycle to read values and perform ALU operations. Similarly, the CACHE also allows for simultaneous instant reads which allow the interfacing of a VGA coprocessor which is currently connected.

The way the VGA works is via the VGA memory controller module. The memory controller will read a value always from address 0 in the CACHE which serves as a pointer to the head of an array in the CACHE. The VGA memory controller will then look at the head of the array and read 3 items which should be saved in this order in sets of 3: x,y coordinate, color values, dimensions.

Each piece of information in the array such as the coordinates are saved as a single word, however are split up into a lower and upper half as the screen resolution only requires 8 bits to access all of the pixels. To form a x,y coordinate word, one would first shift their Y value left by 9 bits, then add the X value to it, and save it into a position in memory where if the pointer pointed to index 0, then the value would be at index i%3=0. For the controller to recognize the end of the array, the terminating word of decimal value 65535 should be saved into the end position (i.e. all 1's), since no coordinate (y being 7 bits wide instead of 8), colour, or dimension could ever result in a word of all 1's.

Seems complicated, but the basic idea is to draw an object, the memory would look like this:

ADDRESS	VALUE
0	0xF7
…	…
0XF7	(Y coordinate, X coordinate)
0XF8	(ASCII char, Colour
0XF9	(width, height)
0XFA	1111111111111111

Unfortunately, in the final CPU project we submitted, we were unable to have the ability to print an ASCII character onto the screen. However, implementation should be very possible. Maybe a project for another future student?

Now you may be wondering how to program the CPU?

In the beginning, you would have to program the CPU yourself by hand, flipping switches for 1 or 0. However that's tedious. Introducing the assembler.

The assembler provides an easy conversion between the assembly for the CPU and machine code that is loaded into the CPU program memory.

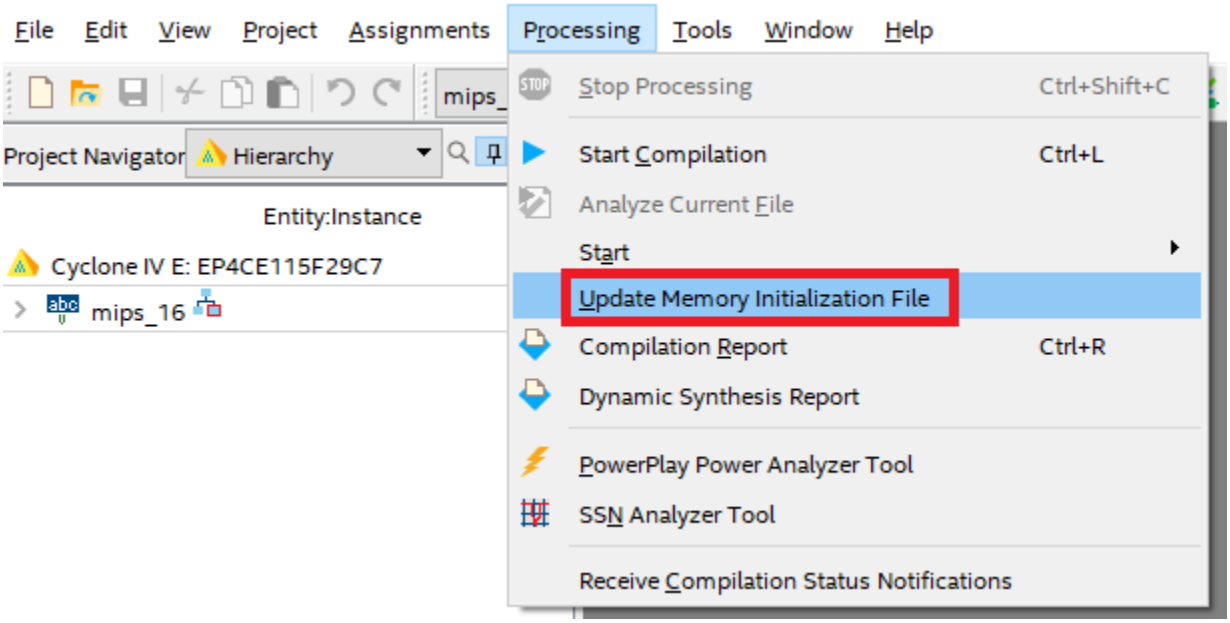
An exe has been included for use on windows machines without python. Other users should use the command: 'python assembler.py' in terminal

If the assembler closes immediately, there was an exception. Either you entered the filename wrong or something really bad happened. Good luck.

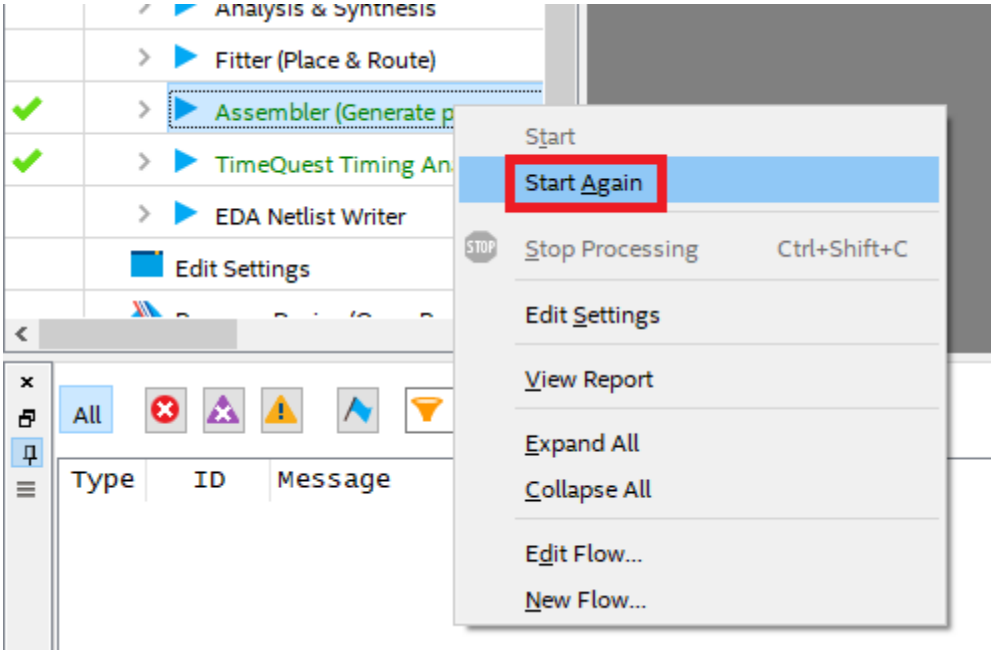
If your .mif file is empty, read the assembler error msg.

The output for the assembler is always titled "PROGMEM.mif", it can be loaded into the CPU by copying the .mif file directly into the folder where the CPU Verilog files are located. To load the memory into the de2 board in Quartus,

click: processing->update memory initialization file.



Then in the left side of the Quartus window, right click on the assembler and click run again.



Then open the de2 programmer and upload to the de2 board.

There are 3 headers that can be specified in the assembler for different types of assembly and conversions.

mips_x16:

This is the lowest level assembly available. Commands are documented in the CTRLUNIT.v. All commands will have either 1 or 2 arguments. Be careful when using register $1 as every operation saves a value to it. $1 is disabled and reserved for the assembler in other modes.

An example program in mips_x16 is fib.txt

mips_x32:

This allows you to use standard MIPS instructions. The assembler then converts them into mips_x16 as a combination of MV, LIM and other commands and then assembles into machine code. registers $1, $30, $31 are disabled and reserved for the assembler. Instead, use the register names $ra, $sp. mips_x16 commands are disabled as they can easily cause undesired results when accidentally moving to reserved registers.

Incase the conversion log is too large and the terminal cannot display all of it, a copy of all the commands converted is saved into command_log.txt

An example program is count.txt

mips_x32+:

This is a free for all mode, commands can be either format however when writing this assembly it is advised to pay special attention to every line you write as creating bugs is extremely easy. The assembler will not check anything except for the bare minimum (if instructions are valid).

Some projects we had in mind:

- Pipelining the CPU and writing a demo program to show how performance is increased
- Writing a game like pong in assembly
- Implementing a text editor with the logic done software side (see the notepad—project)

Congratulations! You now know how to use the mips_16 CPU. Use it well and create something amazing. Build upon it to create new projects that have never yet been done before, Anything is possible. Have fun, and good luck!

See the working version here: https://youtu.be/FNep4bpeVNs

If you have any questions or concerns please don't hesitate to contact us at:

kaiwen.zhu@mail.utoronto.ca

kyles.song@mail.utoronto.ca

Zhu, Kevin and Zhifei, Song