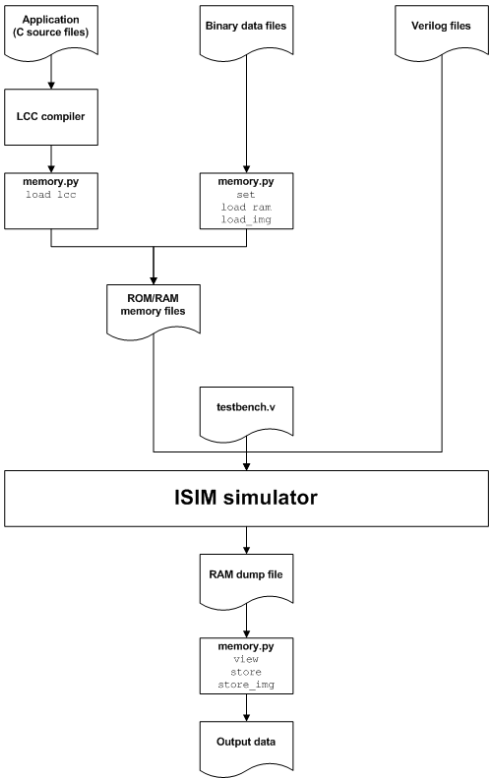
Computation II - 5EIB0
mMIPS implementation and tools

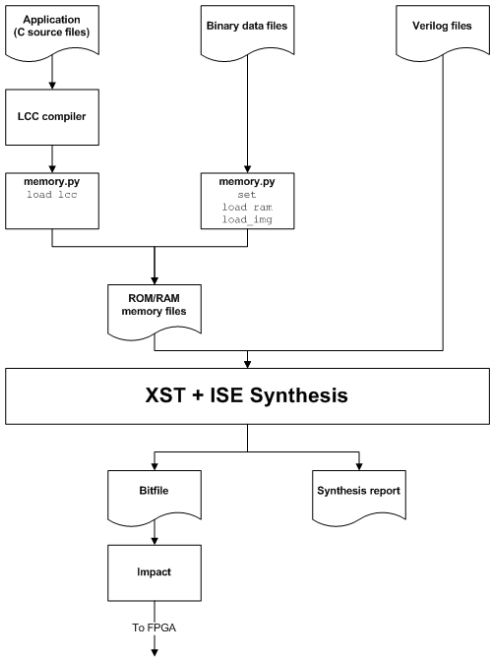
In this documents provides a short introduction on the Verilog implementation of the mMIPS as well as the tools used for simulation and synthesis of this processor. The documents also provides some tips and tricks that might come in handy.

Contents

1 Toolflow

The tool flow for simulation and synthesis of the mMIPS processor are shown below:





2 The mMIPS in Verilog

The mMIPS (pronounced as mini MIPS) is a simplified version of the MIPS processor. Compared to the MIPS it has a reduced instruction set which means that some operations need to be done in software. The following instructions are supported by the mMIPS in hardware:

- addiu, addu, subu
- and, andi, or, ori, xor, xori
- beq, bne
- jal, jalr, jr, j
- lb, lw, sb, sw
- lui
- multu, mfhi, mflo
- slti, sltiu, slt, sltu
- sll, sra, srl (1, 2, 8 bits)

A complete description of all instructions can be found in the `mips_instructions.pdf` file.

Operations that are not directly supported in hardware (e.g., a division) must be performed in software. These operations are called soft operations. Due to the lack of a complete MIPS instruction set in the mMIPS, the processor performs the following operations in software:

- All floating point operations

- Divide, modulo
- Variable distance shifts
- Partial-word operations

2.1 Verilog

The mMIPS model you are working with, is written in Verilog. These files can be found in the `verilog` directory. In this sub-section you can find some more information on the function of some important Verilog modules and files.

- `testbench.v`: instantiates a `mMIPS_sim` module (the complete mMIPS model, including data and instruction memory modules) for simulation purposes only. It writes the `clk`, `rst` and `en` signals (Clock, reset and enable respectively). The testbench will also check if the program has finished and, if so, create a memory dump.
- `mMIPS_sim.v`: connects the mMIPS module to the instruction and data memory.
- `mmips.v`: instantiates and connects all modules in the mMIPS (except the memory modules).
- `mmips_defines.v`: contains some constants that are used in other files.
- `rom.v`, `ram.v`: implement the instruction and data memories respectively. These memories are mapped on dedicated block RAMs on the FPGA in blocks of 2KB. The memories are initialised with the files `mmips/memory/rom/mmips_romXX.hex` and `mmips/memory/ram/mmips_ramXX.hex` respectively.

All other files in the project are fairly self explanatory. Note that some modules (like the registers and muxes) are of variable width (parameter WIDTH). The value of this parameter is defined at instantiation.

2.1.1 Verilog tutorial

In appendix C of the book a detailed tutorial on Verilog is given. A quick tutorial on the basics of Verilog can be found at `http://www.asic-world.com/verilog/verilog_one_day.html`.

2.2 Memory map

The mMIPS uses the following memory map:

Address	Length	field
0x00000000 - 0x00017FFF	96KB	ROM
0x00400000 - 0x0041FFFF	128KB	RAM

2.3 Registers

The mMIPS contains 32 registers. These registers have the following functions:

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and results of a function
v1	3	Expression evaluation and results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

3 Compiling the code

LCC is a retargetable C compiler. The target of a C compiler is the processor for which it generates assembly instructions. The lcc compiler has also been ported to the mMIPS.

3.1 LCC Usage

This section describes how you can use the LCC compiler. As with any other compiler, you must first compile all source code (C/C++) files in object files and then link the object files into a binary.

3.1.1 Compiling to an executable

A source code file (e.g. file.c) is compiled into an binary file (e.g. `mips_mem.bin`) using the following command:

```
lcc file.c -o mips_mem.bin
```

Alternatively, you can compile a (set of) source code file(s) first to a (set of) object file(s) and then link these object file(s) to a binary. In that situation, you should follow the following procedure.

A source code file (e.g. file.c) is compiled into an object file (e.g. file.o) using the following command:

```
lcc -c file.c -o file.o
```

The -c option tells the compiler that it should compile the source code file (e.g. file.c). After the -o option we put the name of the output file produced by the compiler (e.g. file.o).

After all source code files have been compiled, we can link them into one binary. Using the following command, we link the object files file1.o and file2.o into the binary `mips_mem.bin` using the following command:

```
lcc file1.o file2.o -o mips_mem.bin
```

All object files (possibly just one) that are needed to produce the binary must be specified in this command.

3.1.2 Disassembling an executable

You can use the `disas` tool to get a disassemblance (assembler listing) of the binary. For a file `mips_mem.bin`, this is done using the following command:

```
disas mips_mem.bin
```

3.2 LCC with custom MIPS instructions

This section describes how to use custom ops (user defined instructions) that **have already been added to lcc**. The steps used to add these instructions are described below, for your interest. For this we use the following example:

```
int foo(int a, int b)
{
  int res;

  res = ((a << 8) | (b >> 2)) + a - b;

  return res;
}
```

This function must perform a number of operations on the two operands a and b to compute the result. The compiler will expand these operations in at least 5 instructions. However, you as a designer, may have a budget of at most 2 instruction for this operation. So, the operation must be performed faster (use less instructions). The solution for this problem is the use of custom operations (user defined instructions). This requires extra hardware to handle these instructions and the compiler must know how and when to generate these instructions. The following steps describe how the compiler was modified in order to use custom operations. Note that this page does not described what hardware must be added to the MIPS architecture.

3.2.1 Step 1: find opcode space

Find space in the MIPS opcode space for the custom operation that you want to add. See for this Figure A.19 in Appendix A of the book 'Computer Organization and Design'. Note that custom operations can have multiple operands and multiple results. In order to supply all operands and retrieve all results, a custom operation might be implemented by multiple instructions that pass all operands to and from the functional unit where one instructions triggers the functional unit to start the operation.

In our example we have two input operands and one output operand. Thus, we can use one instruction to realize the custom operation that implements our function. We choose to use an R-type instruction, as our instruction fits into this format. The opcode is thus zero. Looking in Appendix A, we see that we can use the function code 0x31, as no instruction currently uses this function code.

3.2.2 Step 2: add patterns to minimips.md

When lcc compiles a program, it generates a tree of the program code. The nodes in this tree represent basic operations like 'add two integers' or 'calculate indirect address from integer value'. The compiler maps these nodes onto the machine instructions. Sometimes, a single node can be mapped onto one instruction, e.g. 'add two integers' can be mapped onto the addu instruction. But other nodes may require multiple machine instructions or one machine instruction may require a sequence of nodes.

Custom operations are generated by patterns (sequence of nodes in the tree) that are very unlikely to occur in normal programs. An example of such a pattern is an integer load of address `0x12344321` whose result is added to another integer and this result is again subtracted from another integer. If the compiler sees this pattern in the tree, then we can instruct it to not output instructions that perform these operations, but instead output our own special instruction. This pattern is specified in the file `lcc/src/minimips.md`, this file also specifies what assembly code to generate for the pattern. This combination of pattern and required output is called a rule.

For our example we could use the following rule:

```
reg: SUBI4(reg, ADDI4(reg, INDIRI4(magic_addr))) \
             "\t.word (%c<<11)|(%0<<21)|(%1<<16>|0x31\n"
```

This pattern has two arguments (reg) which corresponds to the arguments a and b of our custom operation. The result of the custom op is again a value, as is indicated by the `reg` that starts this rule. The `SUBI4` and `ANDI4` keywords indicate nodes in the tree that perform respectively a 4-byte integer subtraction and addition. The `INDIRI4(magic_addr)` indicates a 4-byte integer indirect address calculation with an address equal to the `magic_addr` (`0x12344321`). When this pattern is detected, the compiler emits the `.word` directive which assembles the instruction for the custom op. The destination register (`%c`) and the two source registers (`%0` and `%1`) are inserted at the proper bit positions. Note that the opcode of this instruction is 0 and the function code is `0x31`. Because we generate a `.word` directive instead of a new instruction, we do not have to modify the assembler that translates the assembly code of lcc to binary code.

Note that if a custom operation needs more than two inputs or produces more than one result, we must use multiple instructions. The assembly string contains in that case multiple instructions separated by newline characters (`\n`) to pass all arguments to the functional units. An example of a rule with three inputs and one output is the following:

```
reg: ADDI4(reg, ADDI4(reg, ADDI4(reg, INDIRI(magic_addr)))) \
             "\t.word (%c<<1)|(%0<<21)|0x35\n\t.word (%c<<1)|(%1<<21)|(%2<<16)|0x36\n"
```

This rule is used when three subsequent 4-byte integer additions are found in which the last addition adds an integer to the indirectly calculated magic address. The rule outputs two instructions (both R-type with function codes `0x35` and `0x36`).
To use the newly added custom instructions, lcc must be recompiled. **These steps have been carried out for you, so there is no need to recompile the version of lcc that was provided.**

3.2.3 Step 3: use the custom ops

The easiest way to use a custom operation is to define a macro that expands into the pattern of operations that triggers generation of the custom operation. For our example this corresponds to:

```
#define sfu0(a, b) ((a) - ((b) + *(int *) 0x12344321))
```

Our function looks then as follows:

```
int foo(int a, int b)
{
  int res;

  res = sfu0(a,b)

  return res;
}
```

Note that the program above no longer has the same functional behaviour as compared to the original program. The result of the foo function is different.

You can use the program `disas` to check that our custom operation is really used by the program. The custom operation is visible as an encoded word in the decoded program.

Remark: It is very important that the variables passed to `sfu0` are of the correct type (integer). Else, the compiler will insert type-casts. These destroy the pattern that must be recognized by our rule. As a result, the compiler will not use our custom instruction.

3.2.4 How does it work internally?

Above it is described how you can add a custom operation to lcc. We did not discuss any of the internals of lcc. We simply said that using a pattern which contains a `INDIRI4(magic_addr)` node and some other nodes, we could generate custom operations. You might now be wondering what it this special node `INDIRI4(magic_addr)`. To explain this, we must dive a little deeper into the file `minimips.md`.

The `INDIRI4(magic_addr)` node represents an 4-byte integer indirect address calculation with a node `magic_addr` as input. This `magic_addr` node is also defined in the `minimips.md` file (see line 345) with the following rule: `magic_addr: CNSTP4 "%a" is_magic_addr(a)` The rule says that a `magic_addr` node is a node that contains a constant 4-byte pointer. The value of this pointer is equal to 'a'. The rule says further that this node is only a `magic_addr` node if the `is_magic_addr()` function returns a value zero. This `is_magic_addr` function is also defined in `minimips.md` and is defined as follows:

```
int is_magic_addr (Node p)
{
  return (int) p->syms[0]->u.c.v.p == 0x1234321 ? 0 : LBURG_MAX;
}
```

This function takes the node p, which is possibly our `magic_addr` node, as an input. The function check whether the value (`u.c.v.p`) of the first symbol (`syms[0]`) of this node is equal to `0x1234321`, hence our magic address. If this is the case, zero is returned, else the function returns the `LBURG_MAX` value. This `LBURG_MAX` is equal to the largest possible integer.

4 Making your compiled code ready for simulation

`memory.py` is a tool to create and manage the memory files used for initializing the instruction (ROM) and data (RAM) memories of the mMIPS processor. These memory files are located in `mmips/memory/rom` and `mmips/memory/ram`. The script can be used to insert and extract binary files (your compiled program, images or data), change specific values, view parts of the memory and compare the output for biterrors.

The `memory.py` tool can be found in the `mmips/memory/` directory. The tool is started by opening a terminal, going to this directory (`cd /cygdrive/d/mmips/mmips/memory`) and executing `./memory.py`. The tool gives you a prompt in which you can type commands. To get a list of all available commands, type `help`. To get more info about a specific command, type `help`, followed by the command, e.g. `help load`.

Below some info on the usage of various commands is given:

- `help [cmd]`:
  Print help about `cmd`, if `cmd` is omitted, print a list of all possible commands
- `create [rom|ram|all]`:
  Creates empty memory files. If memory files exist, they are overwritten and cleared. It is needed to create the memory files first, before you can load in data. It can also repair broken memory files. If your simulation does not give the desired output after changing the program code and/or image data, it might be a good idea to clear the memory and reload all instructions/data.
- `load rom|ram|lcc fname [addr]`:
  Load the contents of file `fname` in the rom/ram at address `addr`.
  If option `lcc` is set, the first part of the file will be loaded in the rom, and everything starting from address `0x400000` will be loaded into ram (global variables). `addr` is ignored with lcc parameter. Typical usage for loading the binary file generated by LCC (say `mmips_mem.bin`) would be: `load lcc mmips_mem.bin`.

- `load_img fname [addr] [W H] [w h] [x y]`:
  Load the contents of an .y file `fname` in the ram at address `addr`.
  W and H: width and heigth of the picture.
  w and h: width and height of the crop area.
  x and y: left-top corner of the crop area.
  The default values of W, H, w, h, x and y suffide for the simulation purposes. To quickly simulate, you could try to use a smaller image by setting the w and h parameters to something small. Be aware though, that this might result in incorrect verification when the selected area is very small. Compare the smaller image with a reference output of an image with the same dimensions.

- `store fname [addr] [length]`:
  Store `length` bytes from the ram dump file to binary file `fname`, starting from address `addr`. This converts the hex ASCII output to a binary file.

- `store_img fname [merge_file] [addr] [W H] [w h] [x y]`:
  Store the processed image in the ram dump file to .y file `fname`. If `merge_file` is set, the output will be merged with the contents of this file (replacing the cropped area) and thus shows your processed area within the original image (with a black border). If `merge_file` is set to `none`, the cropped area is inserted into a black image. The W, H, w, h, x and y parameters are the same as the `load_img` command. Use the same values for W, H, w, h, x and y as you used for `load_img`.

- `view rom|ram [from] [length]`:
  Show the contents of the memory file from address `from` to a maximum length of `length`.

- `set rom|ram addr value`:
  Set the memory value of the rom/ram at address `addr` to value `value`. The address should be dword-aligned, meaning that it's a multiple of 4. Value is a 32 bit number and is written in four bytes starting from addres `addr`.

- `dump [fname]`:
  Write the contents of the ram memory files to the dumpfile `fname`. This could only be used to verify that the input has been inserted correctly.

- `compare fname1 [fname2]`:
  Compare the memory dumps `fname1` and `fname2` for the location of the output image. An image size of 32x32 and a start address of 0x2000 is assumed. If `fname2` is ommitted, this defaults to the location of the ram dump file. So to check the latest simulation results against a reference file `mmips/memory/mmips_ram.ref.hex` you can use: `compare ./mmips_ram.ref.hex`.

- `exit`:
  Exit `memory.py`.

5 Viewing the images

`ImProc` can be used to convert .y type images to .png type images. This can be used to visually inspect the output of the filter. The png images can be viewed using the `eog` tool.