Practical Work: Virtual core implementation

Thomas Trouchkine Guillaume Bouffard

October 13, 2021

1 Implementation

The core must be an executable file able to run on an up-to-date 64 bits Debian 11 distribution.

2 Specifications of the core

2.1 The architecture

The core must have a 64 bits architecture composed of at most sixteen internal registers named r_i with $i \in [0, 15]$.

2.2 The binary

The binary must take two mandatory and one optional arguments:

- the code to be executed by the core (mandatory)
- the initial state of the internal registers (mandatory)
- a flag to set or unset the verbose mode (optional)

Usage example: `BIN_NAME <CODE> <STATE> (VERBOSE)`

2.3 The code

The code is a binary file storing the instructions to be executed by the core. The instructions are 32 bits wide and must be stored with big endian organization.

2.4 Internal state file

The internal state file describes the initial values of the different register with the following organization: `REG_NAME=HEX_VALUE`.

Example with the register r_0 initialized to the value `0x23545874`: `R0=0x23545874`.

2.5 The instruction set

2.5.1 Instruction encoding

An instruction is 32 bit wide, the 31^{st} is the most significant bit.

The instruction encoding must be organized as follow:

- Bits 28 to 31: Branch Condition Code (BCC)
- Bits 25 to 27: always set to 0

- Bit 24: Immediate Value (IV) flag
- Bits 20 to 23: Operation Code (opcode)
- Bits 16 to 19: First Operand (ope1)
- Bits 12 to 15: Second Operand (ope2)
- Bits 8 to 11: Destination Register (dest)
- Bits 0 to 7: Immediate Value (IV)

2.5.2 The BCC

The BCC manage how the next value of the Program Counter (PC) is computed. The PC corresponds to the address of the next instruction to be executed. When a branch is happening, the PC is computed from its current value and an offset encoded on the bits 0 to 26 of the instruction. The bit 27 encoded if the offset is positive or negative. Therefore, during a branch, the PC is computed following the equation:

$$PC_{n+1} = PC_n + (-1)^{instr[27]} \times instr[26:0]$$

The different values for the BCC are:

Value	Description	Code
0x8	Unconditional branch	B
0x9	Branch if equal	BEQ
0xa	Branch if not equal	BNE
0xb	Branch if lower or equal	BLE
0xc	Branch if greater or equal	BGE
0xd	Branch if lower	BL
0xe	Branch if greater	BG

For any other value, the PC is computed to target the next instruction to be executed.

2.5.3 The second operand

During the computation, the instruction usually use different operands. From now, the second operand (ope2) refers to:

- The second operand if the 24^{th} bit is set to 0
- The immediate value if the 24^{th} bit is set to 1

2.5.4 The opcode

The opcode encode the different operations the core can do. Its different values are:

Value	Description	Code	Equation
0x0	Logical AND	AND	$dest = ope1$ and $ope2$
0x1	Logical OR	ORR	$dest = ope1$ or $ope2$
0x2	Logical XOR	EOR	$dest = ope1$ xor $ope2$
0x3	Addition	ADD	$dest = ope1 + ope2$
0x4	Addition with carry	ADC	$dest = ope1 + ope2 + carry$
0x5	Comparison	CMP	See section on CMP (2.5.5)
0x6	Subtraction	SUB	$dest = ope1 - ope2$
0x7	Subtraction with carry	SBC	$dest = ope1 - ope2 + carry - 1$
0x8	Move data	MOV	$dest = ope2$
0x9	Logical left shift	LSH	$dest = ope1 << ope2$
0xa	Logical right shift	RSH	$dest = ope1 >> ope2$

2.5.5 CMP opcode

The `CMP` opcode compare the values of ope1 and ope2 and sets the flags used by the branching.

Flag	Set Condition
BEQ	$ope1 = ope2$
BNE	$ope1 \neq ope2$
BLE	$ope1 \leq ope2$
BGE	$ope1 \geq ope2$
BL	$ope1 < ope2$
BG	$ope1 > ope2$

3 Progress of the work

3.1 Base of the core

The first part of the work consists in implementing a working core as described in section 2. The awaited functions are the following:

- `fetch`: read the instruction to be executed from the code and compute the new value of the PC
- `decode`: decode the different parts described in subsubsection 2.5.1 from the instruction
- `execute`: realize the operations described in subsubsection 2.5.4 and in subsubsection 2.5.5

These functions must be tested and the testing program must be provided. An optional verbose mode can be activated to print the following information:

- From the `fetch` function: the BCC value (in hexadecimal), the offset (as a relative integer), the instruction (in hexadecimal), the PC
- From the `decode` function: the opcode, the first operand, the second operand, the destination, the immediate value (all in hexadecimal) and the immediate value flag
- From the `execute` function: the carry, the registers value (in hexadecimal) and the branch flags

3.2 Code compiler

The code compiler's purpose is to translate an assembly description of the code into a binary code. The generated code must match the requirements mentioned in subsection 2.3. The compiler can be developed in any language.

The assembly file must have the `.s` suffix and be organized with the following rules:

- One instruction per line
- An instruction must be organized as follow:
  - `BRANCH_OPCODE OFFSET` in the case of a branch instruction
  - `OPCODE DESTINATION FIRST_OPERAND SECOND_OPERAND` in the case of a data processing instruction

There is no `DESTINATION` with the `CMP` opcode. The difference between a register and an immediate value for the second operand is the presence of the `r` for register.

Examples:

- `B 25` → branch with positive offset of `25` in the code
- `B -3` → branch with negative offset of `3` in the code

- `ADD r1, r2, 4` → $r_1 = r_2 + 4$
- `ADD r1, r2, r4` → $r_1 = r_2 + r_4$
- `CMP r2, 5` → comparison between the register r_2 and the value `5`
- `CMP r2, r5` → comparison between the registers r_2 and r_5

3.3 First program: initialize register values

The first program to be execute on the core is the initialization of the internal registers from a state in which all the registers are initialized to `0`. The program must go from the initial state to the final state presented in the following table.

Register	Initial state	Final state
r_0	`0x0`	`0x0123456789abcdef`
r_1	`0x0`	`0xa5a5a5a5a5a5a5a5`
r_2	`0x0`	`0xaef45d745aff584f`

The program must be stored in a code file named `init_test` and the initial state in a file named `init_state`. An explanation on how the program is built and works is awaited. If a tool is used to make the program it must be mentioned. If the tool is self programmed then it must be provided and will be gratified, any language can be used for the tools.

3.4 Second program: 128 bits addition

The second program to be executed on the core is the addition of two 128 bits integers. The program must compute the result of the following equation:

```
  0x24152dfb45da45dfa521147fde45f45a
+ 0x45dcea451f2d45a4f5554ed4f4522365
= 0x69f2184065078b849a766354d29817bf
```

The initial state for this program must be the following:

Register	Initial state
r_0	`0x24152dfb45da45df`
r_1	`0xa521147fde45f45a`
r_2	`0x45dcea451f2d45a4`
r_3	`0xf5554ed4f4522365`

The result can be stored in any of the registers.

The program you realized must be stored in a code file named `add128_test` and the initial state of the registers in a file named `add128_state`, both files must be provided with your core.

3.5 Third program: 64 to 128 bits left shift

The Third program to be executed on the core is the left shift from a 64 bits integer to a 128 bits integer. The program must compute the result of the following equation:

```
0xf458f452145147de << 0xc = 0xf458f452145147de000
```

The initial state for this program must be the following:

Register	Initial state
r_0	`0xf458f452145147de`
r_1	`0xc`

The result can be stored in any of the registers.

The program you realized must be stored in a code file named `lshift64_128_test` and the initial state in a file named `lshift64_128_state`.

3.5.1 Bonus: 128 bits left shift

Realize a program computing the left shift on a 128 bits integer. The program must be stored in a code file named `lshift128_test` and the initial state in a file named `lshift128_state`.

3.6 Fourth program: 64 bits multiplication

The fourth program to be executed on the core is the multiplication between two 64 bits integers. The program must compute the result of the following equation:

`0xfebc45fe4512695f * 0xf48ef54a = 0xf359b338fe48703f94dc6076`

The initial state for this program must be the following:

Register	Initial state
r_0	`0xfebc45fe4512695f`
r_1	`0xf48ef54a`

The result can be stored in any of the registers.

The program must be stored in a code file named `mul64_test` and the initial state in a file named `mul64_state`.

3.7 Fifth program: 64 to 128 bits factorial

The fifth program to be executed on the core is the computation of the factorial of a 64 bits integer. The results cannot be more than 2^{128}. The program must compute the result of the following equation:

`0x1e! = 0xd13f6370f96865df5dd54000000`

The initial state for this program must be the following:

Register	Initial state
r_0	`0x1e`

The result can be stored in any of the registers.

The program must be stored in a code file named `fac64_128_test` and the initial state in `fac64_128_state`.

3.8 Sixth program: From C code

In this section, the aim is to propose an assembly code which realize the same as the function described in Listing 1.

```
int compare(int A, int B){
  if (A > B)
    return 0xfe;
  else
    return 0xaf;
}
```

Listing 1: Compare function

The assembly code must be in the report with an explanation of why it corresponds to the given C code. Also, programs using this code with different values for `A` and `B` are awaited.

4 Questions

1. Which parts of a 64 bits processor are 64 bits wide ?
2. Which instructions can potentially create a carry ?
3. What is the purpose of the add carry (`ADC`) instruction ?
4. What are the check to realize during a branch instruction ?
5. Is it possible to pipeline the virtual core ?

5 Expected deliverable

The deliverable must be composed of:

- The source code (well commented, with its documentation and a `Makefile`) of the core
- The code compiler and its documentation
- The programs
- A 3 pages report (10pt font size) including the explanation of the implementation, a quick information on how to use it, the explanation of the tested program (especially those who do not work) and the answers to the questions.

The deliverable must be sent as an archive at both contact@bouffard.info and thomas@trouchkine.com until the 17^{th} of December, 2021 at 23:59 Paris localtime. A return receipt will be sent later to acknowledge the mail reception. Every hour late after late midnight decreases your rating.

Group is allowed up to 2 peoples.