Anthony Debruyn
Quentin Delhaye
Alexis Lefebvre
Aurélien Plisnier

Labo 1 : L'amour du RiSC16

Dragomir Milojevic & Yannick Allard $\boxed{\text{ELEC-H-473}}$

Question 1

The code implements a Fibonacci sequence.

addi 2,0,1	Add 1 to the content of R0 and store the result into R2. (1 is put into R2) Increments PC.
sw 2,1,0	Store the content of R2 in the Data Mem at the address found in R1 with an offset of 0. (1 is put into address 0)
addi 1,1,1	Add 1 to the content of R1 and store the result into R1. (1 is put into R1)
sw 2,1,0	Store the content of R2 in the Data Mem at the address found in R1 with an offset of 0. (1 is put into address 1)
addi 1,1,1	Add 1 to the content of R1 and store the result into R1. (2 is put into R1)
add 3,2,2	Add the content of R2 to the content of R2 and stores the result into R3 (2 is put into R3).
sw 3,1,0	Store the content of R3 in the Data Mem at the address found in R1 with an offset of 0. (2 is put into address 2)
addi 1,1,1	Increments the content of R1.
addi 7,0,7	Put 7 into R7.
beq 7,0,end	If the content of R7 is 0, jump to label end, else increment PC.
lw 2,1,-2	Load a word from the data memory in R2. The word to load is at the address contained in R1 minus 2.
add 3,3,2	Add the content of R2 to the content of R3 and stores the result into R3.
sw 3,1,0	Store the content of R3 in the Data Mem at the address found in R1 with an offset of 0.
addi 1,1,1	Add 1 to the content of R1 and store the result into R1.
addi 7,7,-1	Decrement the content of R7 of 1.
beq 0,0,loop	Inconditionnal jump to the label loop.

Question 2

See question 1.

Question 3

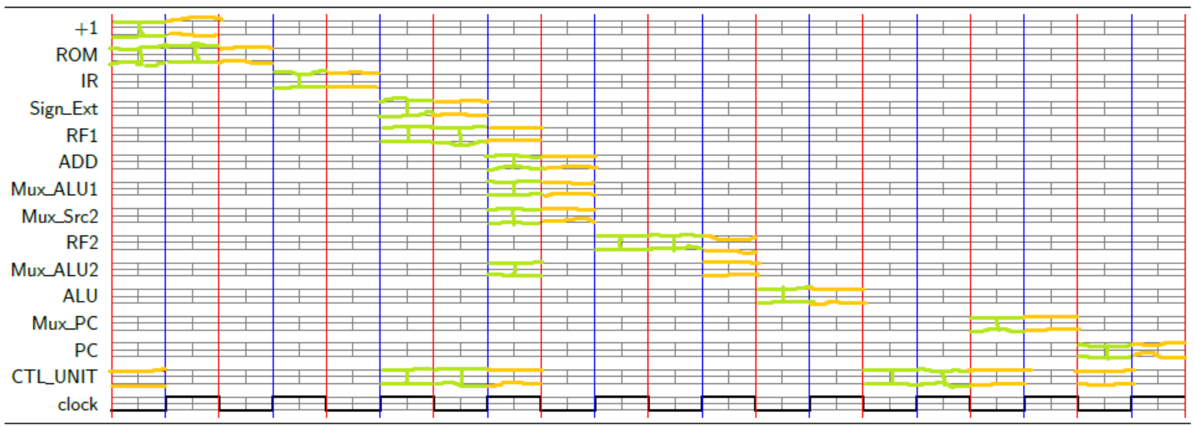

Figure 3: BEQ chronogram

Question 4

```
//R1 : op 1
//R2 : op 2
//R3 : mask, then final result
//R4 : report flag
//R5 : temp 1, used fo R1 NAND R3, loop verification, temporary serious
    science, final temp to check flag status
//R6 : temp 2, temp 0xFFFF for serious science
movi 1,0xEFFF //We want to store a 16-bits word. "addi" is not enough. op1
movi 2,0xEFFF // op2
addi 3,0,1 //set the mask in R3, initialized at 1
nand 5,1,3 //NOT(R1 && R3) in R5
nand 6,2,3
beq 5,6,flag //If equal, maybe set the flag, iff the result is not 0xFFFF
beq 0,0,loop //If not equal, skip the operation setting the flag (label "flag")
    and go directly to the loop.
flag: movi 5,0xFFFF //put 0xFFFF inside R5. R5 will act as a buffer (its
    content does not interest us anymore)
beq 5,6,loop //If R6 (the result of R2 NAND mask, which was equal to R1 NAND
    mask) == 0xFFFF, it means that at the set bit in the mask (the position of
    the bit set to 1 in the mask), we had 0 in both operands (stored in R1 and
    R2). That means that we would have the same NAND mask, but no report needed.
add 4,3,3 //Set the mask at the mask + the mask (mask stored in R3). Flag.
loop: movi 5,0x4000 //Stores the constant to be checked.
beq 3,5,final_op //We want to run the loop 14 times, that is, until the mask is
    equal to 0100 0000 0000 0000 (0X4000 in hexa). We'll increment the mask at
    each run and it will eventually end up on 0x4000.
add 3,3,3 //Shift the mask. We want to move the set bit one place to the left,
    this is done by mutlplying the mask by 2.
nand 5,1,3
nand 6,2,3
beq 5,6,flag_loop
nand 5,4,3 //If not equal proceed to some serious science. Temp(R5) = flag NAND
    mask
nand 5,5,3 //Temp = temp NAND mask
movi 6,0xFFFF
```

```
beq 5,6,flag_loop //If temp == 0xFFFF, set flag.
beq 0,0,loop //If not equal, just ignore the flag and rerun.
flag_loop: movi 5,0xFFFF //May be made more efficient by using R7
beq 5,6,loop
add 4,3,3
beq 0,0,loop
//Final operations
final_op: add 3,3,3 //Update mask
nand 5,1,3
nand 6,2,3
beq 5,6, report
nand 5,4,3 //If not equal proceed to some serious science. Temp = flag NAND mask
nand 5,5,3 //Temp = temp NAND mask
movi 6,0xFFFF
beq 5,6,report //If temp == 0xFFFF, report.
beq 0,0,end
report: movi 5,0xFFFF
beq 5,6,end
addi 4,0,1 //Carry bit.
end: add 3,1,2 //Addition
addi 5,0,1 //Set R5, temp, to check the flag
beq 5,4,halt
addi 4,0,0 //If the msb of the flag array is not set, this means the operation
    does not need a report => empty the array
beq 0,0,halt
halt: halt
```

Question 5

```
addition_report: nop
[code from question 4]
halt: beq 0,7,0
```

Question 6

In general, a synchronous circuit can be made from an asynchronous one by adding bistables between combinatorial blocks and a clock to synchronize those bistables. It increases the calculus rate but also the time taken for a particular input to produce an output.
The surface of the processor will increase to allow for the housing of all those new elements. The distance between different combinatorial blocks has to be minimized to reduce the problem with critical races.

"The majority of the blocks run in an asynchronous manner. It means that between blocks, there is no intermediate register controlled by a clock or a control signal. On the other hand, the control unit is synchronized with the clock, and, the program counter, the register bank and the data memory are synchronized with control signals for the writeback operations. Globally, synchronization is achieved at the instruction level because the read control signal of the program memory is produced by the clockdriven control unit." (RISC 16, simulators)

The internal propagation is asynchronous, to make it synchronous synchronisation bistables should be added.

Question 7

Here are some things we thought would be interesting on the sofware side:

- A more complete set of bitwise instructions, like OR, XOR, AND, NOT, XNOR.
- A multiply/divide instruction.
- The unary minus operator, or the binary minus operator.
- Comparaison operations, like greater than, lower than...

On the hardware side:

- A multiplier for the multiplications.
- A pipeline design for the processor to allow a better calculation rate.
- An interruption system to deal with important procedures.

Question 8

Here is the idea of the algorithm:

```
// R : current result
// A : operand A
// B : operand B

counter = 0
B' = B

loop:
        if (A.currentBit == 1){
                R = R + B'
    }

    B' = B' + B'

    if (counter == 16)
        goto write_res
    else{
        A.currentBit -> nextBit
        counter++
    }
    goto loop

write_res:
        put the result in R3 & R4

}
```

The idea of the algorithm is to do like the written binary multiplication. At the beginning of the code, we choose one of the 2 operands, let's say A (the other is B). A and B are encoded on 16 bits. We loop on the 16 bits of A. For each bit, if it is set to 1, we add the current B′ to the current result R. At each iteration, we multiply B′ by 2, and increment the counter. When the counter is equal to 16, we get out the loop and write the result in the 2 registers.

The code:

```
// M0 : A
// M1 : B LSB
// M2 : B MSB
// M3 : mask
// M4 : result LSB
// M5 : result MSB
// M6 : counter
MOVI 1, 2 // Put operant A into R1
MOVI 2, 0xFFFF // Put operant B into R2
SW 1, 0, 0
SW 2, 0, 1
ADDI 3, 0, 1
SW 3, 0, 3

loop_prim: LW 2, 0, 3 // Load mask
LW 1, 0, 0 // Load A into R1
NAND 3, 1, 2 // NAND of mask & A
MOVI 4, 0xFFFF // AND between R1 & R2 == 00...0 if A.curBit == 0 => NAND ==
    11...1
BEQ 3, 4, not_one

// If curBit == 1, do R = R + B (if curBit == 0, we do nothing)

LW 1, 0, 1 // B LSB in R1
LW 2, 0, 4 // R LSB in R2
ADDI 3, 0, 45 // Prepare the call to addition_report
JALR 7, 3 // Jump with save of return address to R7

SW 3, 0, 4 // New result LSB from R3 stored in M4
LW 5, 0, 5 // Result MSB loaded in R5
ADD 5, 4, 5 // Add R4 (carry bit) & R5 in R5
LW 6, 0, 2 // Load B MSB to R6
ADD 5, 5, 6 // Add B MSB & result MSB
SW 5, 0, 5 // Save new R MSB in M5

// end



// Do B = B + B
not_one: LW 1, 0, 1 // B LSB in R1
LW 2, 0, 1 // B LSB in R2
ADDI 3, 0, 45 // Prepare the call to addition_report
JALR 7, 3 // Jump with save of return address to R7

SW 3, 0, 1 // Save new B LSB to M1
LW 5, 0, 2 // Load B MSB to R5
ADD 5, 5, 5 // Add B MSB & B MSB and write to R5
ADD 5, 4, 5 // Add the carry bit if any
SW 5, 0, 2 // Save the new B MSB to M2

// End of B = B + B
// Load counter and increment
LW 1, 0, 6
ADDI 1, 1, 1

// Check if counter == 16
ADDI 2, 0, 16 // Load 16 into R2
BEQ 1, 2, write_res // If last bit, go to write result
// Else
SW 1, 0, 6 // Save counter
LW 1, 0, 3 // Load mask
ADD 1, 1, 1 // Mask x2
SW 1, 0, 3 // Save mask

```

```
BEQ 0, 0, loop_prim // Unconditional jump to beginning of loop

write_res: LW 3, 0, 4 // Load result LSB into R3
LW 4, 0, 5 // Load result MSB into R4
halt

// Put the LSB of R1 + R2 in R3 with carry bit in R4
addition_report: nop
//R1 : op 1
//R2 : op 2
//R3 : mask, then final result
//R4 : report flag
//R5 : temp 1, used fo R1 NAND R3, loop verification, temporary serious science, final temp to check flag status
//R6 : temp 2, temp 0xFFFF for serious science
addi 3,0,1 //set the mask in R3, initialized at 1
nand 5,1,3 //NOT(R1 && R3) in R5
nand 6,2,3
beq 5,6,flag //If equal, maybe set the flag, iff the result is not 0xFFFF
beq 0,0,loop //If not equal, skip the operation setting the flag (label "flag") and go directly to the loop.
flag: movi 5,0xFFFF //put 0xFFFF inside R5. R5 will act as a buffer (its content does not interest us anymore)
beq 5,6,loop //If R6 (the result of R2 NAND mask, which was equal to R1 NAND mask) == 0xFFFF, it means that at the set bit in the mask (the position of the bit set to 1 in the mask), we had 0 in both operands (stored in R1 and R2). That means that we would have the same NAND mask, but no report needed.
add 4,3,3 //Set the mask at the mask + the mask (mask stored in R3). Flag.
loop: movi 5,0x4000 //Stores the constant to be checked.
beq 3,5,final_op //We want to run the loop 14 times, that is, until the mask is equal to 0100 0000 0000 0000 (0X4000 in hexa). We'll increment the mask at each run and it will eventually end up on 0x4000.
add 3,3,3 //Shift the mask. We want to move the set bit one place to the left, this is done by mutlplying the mask by 2.
nand 5,1,3
nand 6,2,3
beq 5,6,flag_loop
nand 5,4,3 //If not equal proceed to some serious science. Temp(R5) = flag NAND mask
nand 5,5,3 //Temp = temp NAND mask
movi 6,0xFFFF
beq 5,6,flag_loop //If temp == 0xFFFF, set flag.
beq 0,0,loop //If not equal, just ignore the flag and rerun.
flag_loop: movi 5,0xFFFF //May be made more efficient by using R7
beq 5,6,loop
add 4,3,3
beq 0,0,loop
//Final operations
final_op: add 3,3,3 //Update mask
nand 5,1,3
nand 6,2,3
beq 5,6, report
nand 5,4,3 //If not equal proceed to some serious science. Temp = flag NAND mask
nand 5,5,3 //Temp = temp NAND mask
movi 6,0xFFFF
beq 5,6,report //If temp == 0xFFFF, report.
beq 0,0,end
report: movi 5,0xFFFF
beq 5,6,end
```

```
addi 4,0,1 //Carry bit.
end: add 3,1,2 //Addition
addi 5,0,1 //Set R5, temp, to check the flag
beq 5,4,halt
addi 4,0,0 //If the msb of the flag array is not set, this means the operation
    does not need a report => empty the array
beq 0,0,halt
halt: JALR 0,7
```

Since the number of calls to the addition function is function of the number of bits set to 1 in the A operand, the number of cycles needed for the program to compute the value depends on it too. It is thus best to use the number with the least number of bits set to 1 as the A operand. One possible optimisation of the code would be to select the best operand as A before the present code is launched.

The largest number of instructions needed is 6258, when the A operand is 0xFFFF. The lowest is 3013, for A = 0. We thus see that multiplication is very time consuming and ressource consuming. An integrated multiplier would be better, to allow the use of a multiplication instruction.

The url of the latex document, in case you want to copy the code: `https://www.writelatex.com/read/vzxmttrgsfhr`.