The Moncky processor family

Kris Demuynck

17th August 2022

Abstract

This document describes the architecture and implementation of the Moncky processor family. It is assumed that the reader is familiar with logic circuits, processors, and Verilog. This document is a work in progress and will be updated regularly. The home of the Moncky project is

https://hackaday.io/project/181269-the-moncky-project

and

https://gitlab.com/big-bat/moncky

for all source code.

Copyright

Kris Demuynck

The Moncky processor family

Contents

Chapter 1

Introduction

The Moncky-1 processor was originally conceived to accommodate students in learning and experimenting with processors and assembly language. It is a 16-bit processor that has a single cycle RISC architecture with a minimal instruction set (only 7 instructions). It does not have interrupts and has a Harvard architecture. This processor cannot be implemented in an FPGA as it contains busses and 3-state buffers (which are not available on an FPGA). It is implemented as a simulation in Logisim (http://www.cburch.com/logisim/) and Digital (https://github.com/hneemann/Digital). The course text that describes logic circuits and the Moncky-1 processor can be found at:

`https://drive.google.com/file/d/1ngHiIrxpOfb0dO7pm3ZOe7q8XEXCb-b6/view?usp=sharing`

The Moncky-2 processor is a synthesizable version of the Moncky-1. The architecture was changed significantly with extra instructions in mind (these are available in the Moncky-3 processor). There is a simulation available in Digital.

The Moncky-3 processor is an extension of the Moncky-1 and 2 instruction set. It still executes every instruction in one clock cycle but has instructions that can do more things at once. The Moncky-3 processor also has an interrupt line so external hardware can send an interrupt which will stop the processor and make it execute a specific interrupt handler.

All this is explained in this document.

The Moncky-3 computer is a full blown computer containing the Moncky-3 processor, memory, ROM, video output, keyboard input, and SPI. It is described in a separate document on gitlab.

Apart from the hardware, there is also software:

- The Moncky assembler can translate Moncky assembly language to machine code. It supports all Moncky processors.
- The Moncky compiler compiles programs written in the Moncky programming language into assembly.
- The Moncky simulator simulates the Moncky computer including screen and keyboard

These programs are all written in java and can also be found on gitlab. The description is put in separate documents, also on gitlab.

The home page for the Moncky project is:

https://hackaday.io/project/181269-the-moncky-project

All code and documentation can be found in gitlab:

https://gitlab.com/big-bat/moncky

Chapter 2

The Moncky-1 processor

In this chapter a complete working processor (the 'Moncky-1') is presented. It is a fictive processor that is specifically designed for educational purposes. Although real processors are much more complicated it is capable of performing the same tasks, except for tasks that involve interrupts. It is perfectly feasible to create a physical chip for the Moncky-1, using ASIC technology. Creating an implementation on an FPGA is not possible as there are multiple busses and tri-state buffers which are not available in most FPGA's (see also the Moncky-2 processor). A simulation in Logisim and Digital exists and can be found at gitlab:

`https://gitlab.com/big-bat/moncky`

The Moncky-1 processor has the following properties:

- it has a 16 bit architecture: all registers and all instructions are 16 bits wide
- there are only 7 instructions
- each instruction can be performed in one clock cycle
- memory is limited to 2x128 KiB
- there are no interrupts
- it has a Harvard architecture

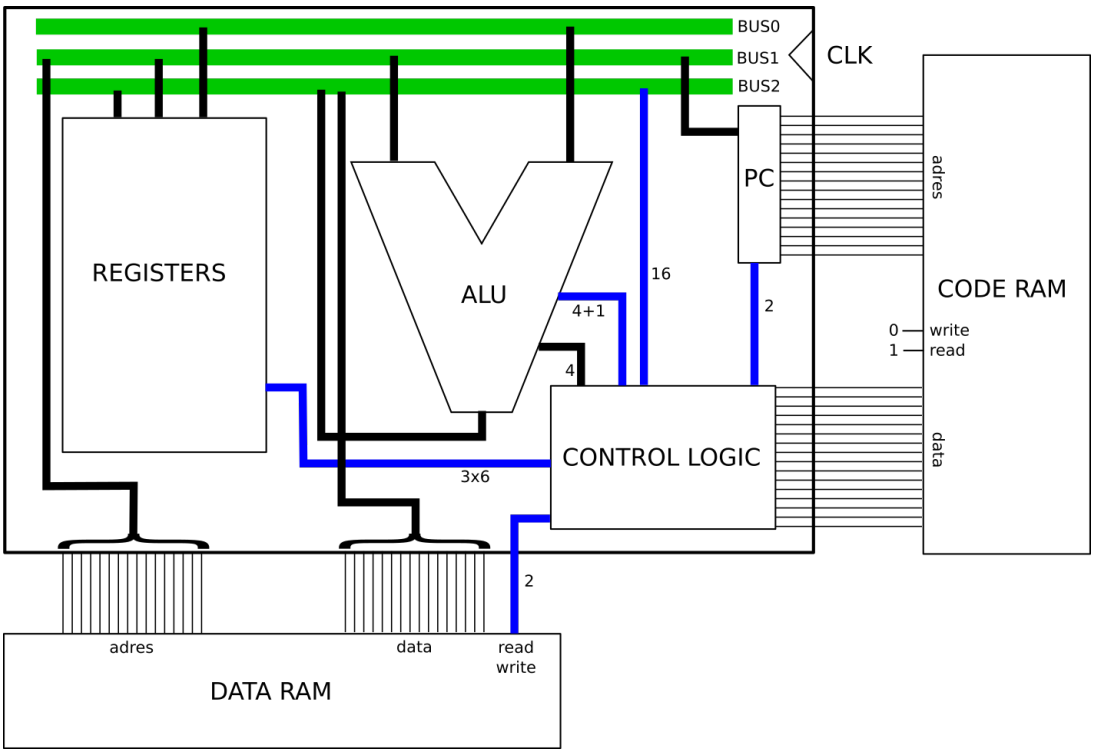
2.1 Architecture

The Moncky-1 processor contains the following components:

- the 'registers': these hold temporary data. There are 16 registers which are all 16 bits wide.
- the 'PC' register which contains the address of the current instruction
- the 'ALU': this will perform arithmetic and logic operations on the data stored in the registers
- the 'instruction decoder' or 'control logic': this interprets the instructions and controls the other components in the processor

- internal busses: these connect all the components together

The Moncky-1 processor has the following architecture:



The black lines in this diagram are data lines and the blue ones are control lines.

Two RAMs are connected to the processor (Harvard architecture). Both RAMs have an address bus that is 16 bits wide. The data bus is also 16 bits wide. A RAM can thus be regarded as an array of 65536 values. Each value is a 16 bit number. The RAMs are assumed to be built with D-latches and to be asynchronous which means they don't have a clock. Reading from the RAM is done continuously as the address bus is directly connected to the MUX in the RAM.

The 'code RAM' contains instructions that need to be executed. Each instruction is 16 bits wide. The 'data RAM' is a memory that will hold the values of variables. Using two RAMs makes the design of the processor simple and keeps performance high.

There are 16 registers in the Moncky-1. Each register can hold a number of 16 bits. The registers are numbered: r0, r1, ..., r15. They are used to hold temporary data i.e. data that is used to perform a calculation. If data needs to be stored, it is always saved to the data RAM.

The program counter (PC) is a separate register directly connected to the address bus of the code RAM. It is a counter that starts at 0 and increments each clock cycle. It holds the address of the instruction that needs to be executed. The code RAM is hardwired for reading. It will put the value at the given address on its data bus. The register PC is sometimes called 'instruction pointer' (IP) as it points to the current instruction. In practice the code RAM can be implemented as an EEPROM that is programmed by external hardware.

The ALU (Arithmetic Logic Unit) is capable of doing calculations. It reads two numbers, performs the calculation, and puts the result on its output. It also generates 'flags' that give information about the result. The inputs are on top and the output is on the bottom.

There are 3 busses in the Moncky-1. They allow the components to send information to each other. For example, the ALU will read the two numbers from busses 0 and 1. The value on bus

0 and bus 1 will come from a chosen register. The control logic will determine which registers are put on the busses. The result of the ALU is put on bus 2 and can be read into a register.

The control logic is connected to the data bus of the code RAM. It will read the instruction and will translate this into control signals to the other components.

The processor has a clock input. All registers, including PC, share are connected to it. They respond to the falling edge of the clock.

2.2 The instruction set

This section describes the instruction set architecture (ISA) of the Moncky-1 processor.

In the Moncky-1 processor all instructions are exactly 16 bits long. It knows the following 7 instructions:

- stop executing instructions (this is used to stop the processor after executing a program)
- load a number (a constant) into a register
- perform a calculation with the registers and put the result back into a register
- load a number from data RAM (at a given location) into a register
- store a number from a register into data RAM at a given location
- jump to a specific address in the code RAM
- jump to a specific address in the code RAM if the result of the previous calculation was zero, negative, positive, ...

Every instruction is embedded in 16 bits. These bit patterns constitute the 'machine language' of the processor. Because they are very hard to read, a more human readable version is also provided. It consists of a short command name (opcode) and parameters (operands). This language is called the Moncky-1 'assembly language'.

In the Moncky-1 processor the 7 instructions are encoded in 16 bits as follows:

machine language	meaning	example
`0000 xxxx xxxx xxxx`	the processor stops	halt
`00x1 iiii iiii rrrr`	R[r] = 0000 0000 iiii iiii	li r3, 42
`01xx AAAA rrrr ssss`	R[r] = R[r] ALU R[s]	add r3, r4
`100x xxxx rrrr ssss`	R[r] = RAM[s]	ld r2, (r1)
`101x xxxx rrrr ssss`	RAM[s] = R[r]	st r1, (r2)
`110x xxxx xxxx rrrr`	PC = R[r]	jp r12
`1111 xxxx xccc rrrr`	if (condition) PC=R[r]	jpz r1

The following notation is used:

- R represents the array of registers
- 'iiii iiii' stands for an 8-bit value that is embedded into the instruction. This is also called an 'immediate' value as the value is immediately available from the instruction.
- "rrrr" and "ssss" stand for register numbers (0-15)

- "AAAA" stands for an ALU operation. The following operations are supported:
  - 0000 = NOP (No OPeration: the result is equal to the second input of the ALU)
  - 0001 = OR (a bitwise OR)
  - 0010 = AND (a bitwise AND)
  - 0011 = XOR (a bitwise XOR)
  - 0100 = ADD (an addition)
  - 0101 = SUB (a subtraction)
  - 0110 = SHL (shift left: the first number is shifted to the left. The number of bits to shift is indicated by the second number)
  - 0111 = SHR (shift right)
  - 1000 = ASHR (arithmetic shift right)
  - 1001 = NOT (logic inverse: the result is the 1-complement of the second input)
  - 1010 = NEG (negative: the result is the 2-complement of the second input)
- "ccc" indicates a certain condition. Every time a calculation is done by the ALU, 4 flags are set according to the result: the carry flag, the zero flag, the sign flag, and the overflow flag). These flags can be used to decide to jump to a certain location in the code RAM. The following conditions can be checked:
  - 000 = 'c' = if the carry flag is equal to 1 (the calculation resulted in 1 extra bit being discarded)
  - 001 = 'nc' = if the carry flag is equal to 0 (the calculation did not result in an extra bit)
  - 010 = 'z' = if the zero flag is equal to 1 (the result of the calculation was 0)
  - 011 = 'nz' = if the zero flag is equal to 0 (the result of the calculation was not equal to 0)
  - 100 = 's' = if the sign flag is equal to 1 (the result of the calculation was negative)
  - 101 = 'ns' = if the sign flag is equal to 0 (the result of the calculation was not negative)
  - 110 = 'o' = if the overflow flag is equal to 1 (the calculation resulted in an overflow)
  - 111 = 'no' = if the overflow flag is equal to 0 (the calculation did not result in an overflow)
- "x" stands for don't care (the values of these bits can be anything; we will always put them to zero in the examples below)

The 4 most significant bits indicate which instruction is to be executed. They are called the 'opcode'. The other bits hold the parameters for that instruction. These are called the 'operands'.

A small program for the Moncky-1 could look like the following:

```
;machine code         assembly    meaning
0001 0000 0101 0000   li r0, 5    ; load the number 5 in register 0
0001 0000 0011 0001   li r1, 3    ; load the number 3 in register 1
0100 0100 0000 0001   add r0, r1  ; add register 0 and 1 together
                                    and put the result in register 0
0001 0000 0010 0001   li r1, 2    ; load the number 2 in register 1
1010 0000 0000 0001   st r0, (r1) ; store the value van r0 (8 in this case)
                                  ; at location given by r1 (2 in this case)
                                  ; in the data RAM
0000 0000 0000 0000   halt        ; end the program
```

The left column shows the binary form of the machine code. These are the bits that are stored in the code RAM. The middle column shows the assembly version of the instructions. The third column contains comments (here a comment starts with a semicolon).

This program is stored in the code RAM. This means the code RAM will contain the following values (in hexadecimal): 1050, 1031, 4401, 1021, A001, 0000.

When the processor is started, PC is set to 0. This means that the first instruction will be executed. After one clock cycle PC is incremented and the next instruction can be executed. This continues until PC=5. Since the instruction is 'halt', PC will not be incremented any more. This halts the processor.

2.3 The internal components

In this section the different components of the Moncky-1 processor are discussed in more detail. There is a simulation in Logisim and Digital available via:

```
https://gitlab.com/big-bat/moncky
```

2.3.1 The busses

The Moncky-1 contains 3 busses that connect the data inputs and outputs of the different components. They are numbered 0, 1, and 2. Every bus consists of 16 lines. The registers, the ALU, the PC register, the data RAM and the control logic are connected to them using tri-state buffers. The control logic is the master of the busses and decides who can write to which bus.

2.3.2 The registers

The Moncky-1 processor contains 16 registers of which each represents a 16-bit number. All registers are built with master-slave D flip-flops. They all share the same clock. At the falling edge of each clock cycle a register can read a new value from a bus of write its value to a bus. MUX's and DEMUX's are used to select the right register and to indicate if it needs to read, write, or just retain its value.

Looking at the registers as one big sequential circuit, it has the following connections:

- a clock input that is forwarded to all 256 flip-flops.
- per bus:
  - 16 connections that can be configured as input or output. These are used to send or retrieve data
  - 4 inputs that determine which register should be connected to the bus (they lead to the MUX's and DEMUX's). These come from the control logic.
  - 1 input (rw) that determines if the the selected register should read or write. Its value is determined by the control logic.
  - 1 input (enable) that connects the selected register to the bus, or disconnects it. This will drive the tri-state buffers. Its value is determined by the control logic.

For example, if the control logic wants to send the information on bus 0 to register r5, the 4 inputs for bus 0 will be put on 0101. The rw input will be 0 (read) and the enable input will be 1. The enable inputs of the other busses will be set to 0 and their rw will be put at 1 (write). When the clock now goes from 1 to 0, register r5 will read the value from bus 0.

In fact the registers form a triple port RAM of 16 words. Each port is connected to a bus.

The PC register is also a register, but it can also increase its value by 1 every clock cycle. The PC register can put its value on bus 1 or read a new value from it.

2.3.3 The ALU

The ALU is in fact a very big combinational circuit. It has 37 inputs and 20 outputs. The output is always directly determined by the input. The inputs are:

- 16 inputs for the first number
- 16 inputs for the second number
- 1 input that determines if the output of the ALU should be connected to bus 2
- 4 inputs to select the operation the ALU needs to perform.

The outputs are:

- 16 outputs for the result
- 4 outputs that give information about the result (the flags):
  - zero flag: is the result equal to 0?
  - carry flag: is the most significant carry bit set?
  - sign flag: is the result negative?
  - overflow flag: was there a 2-complement overflow?

2.3.4 The control logic

The control logic is a main component in the processor. It interprets an instruction and drives all other components. It is a big combinational circuit with 20 inputs and 42 outputs.

The inputs are:

- 16 bits for the instruction. These bits come from the code RAM.
- 4 bits coming from the ALU (the flags)

The outputs are:

- 2 signals go to the control bus of the data RAM and indicate if the RAM needs to read or write or do nothing
- 4 signals go to the ALU to indicate the operation it needs to perform
- 1 signal goes to the ALU to indicate if it has to put its output on bus 2

- 2 signals to the PC register to determine if the register should increment, read its value from bus 1, or write its value to bus 1
- 3x6 signals to the registers determining which register should be connected to each bus and what action it should take
- 16 data signals to bus 2 to send a number directly to a register

The control logic consists of 7 circuits that drive their respective instruction. A MUX is then used to select the right control signals based on the opcode of the instruction.

The Moncky-1 processor is designed so that every instruction in executed in one clock cycle. When the clock is low, the control logic interprets the instruction and puts all the control signals to the right values. When the clock goes high nothing changes. The ALU and the data RAM get time to perform their operation and will put the result on the right bus. When the clock goes from one to zero (falling edge), the registers store the new value and the program counter is increased. When PC changes, a new instruction is presented to the control logic. The next subsections will discuss a few instructions and explain in detail how the control logic drives all components in order to execute the instruction.

2.3.4.1 Load instruction (ld)

The first instruction that will be discussed is 'ld r2, (r1)'. In this instruction r1 points to a location in the data RAM. The processor needs to load the value at that location and store it in register r2. Let's assume that register r1 contains the value 1024 and that the instruction itself is stored at location 2040 in the code RAM.

At the falling edge, when the clock goes from 1 to 0, PC reaches the value 2040 (`0000 0111 1111 1000`). This value is put on the address bus of the code RAM which drives the MUX within it, selecting the right value at this location. The RAM will respond by putting the instruction (`1000 0000 0010 0001`) on its data bus. The control logic notices this instruction begins with '`100`', so it knows it is a 'ld' instruction. The following signals are then sent to all components:

- the registers get the following signals per bus:
  - bus0: is decoupled (enable signal is put to 0)
  - bus1: bits 0-3 of the instruction (`0001`) are forwarded so that register r1 is selected. The rw signal is put to 0 (read) and the enable signal is put to 1. As a result, register r1 will put its value on bus1
  - bus2: bits 4-7 of the instruction (`0010`) are forwarded so that register r2 is selected. The rw signal is put to 1 (write) and the enable signal is put to 1. As a result, register 2 will read its value from bus2
- the data RAM is configured for reading (read=1, write=0). It will put the data on bus2.
- register PC is asked to increment itself on the next clock cycle

As a consequence of these signals, the value of register r1 (1024) flows through bus1 to the address bus of the data RAM. The data RAM will put the value stored at this location on bus2. The value is read by register r2. This register contains flip-flops so it does not yet store the new value.

When the clock goes from 1 to 0, register 2 will store the new value and PC will increment. The instruction is now executed and, since PC is incremented, the next instruction will appear in the control logic.

2.3.4.2 A calculation (add)

The second instruction is 'add r2, r3'. This instruction adds the values of r2 and r3 and puts the result back into r2. The instruction looks like: `0100 0100 0010 0011`.

When the control logic gets this instruction, it notices it begins with '01'. The following signals are now sent:

- bits 8-11 of the instruction contain the operation that needs to be performed ('add' in this case). They are sent to the ALU.
- the ALU is instructed to put its value on bus2 (using tri-state buffers)
- the registers get the following signals:
  - bus0: register r2 puts its value on this bus
  - bus1: register r3 puts its value on this bus
  - bus2: register r2 reads its value on this bus
- register PC is instructed to increment

Notice that register r2 puts its value on bus0 and reads a new value from bus2. This is possible due to the master-slave architecture of the flip-flops.

The values of r2 and r3 flow through the busses to the ALU. The ALU performs the right operation (it adds them together) and puts the result on bus2. This result is read by register r2.

When the clock now goes to 0, the sum of r2 and r3 is stored in r2 and PC is incremented.

2.3.4.3 A conditional jump (jpz)

The last instruction is 'jpz r15'. This instructs the processor to jump to another location in the code RAM (given by the value of r15), but only if the 'zero flag' is set (meaning that the result of the last calculation was zero). The instruction looks like: `1111 0000 0010 1111`.

The control logic will notice that the instruction begins with 1111. It will therefore inspect bits 4-6 from the instruction. As these bits are equal to 010, the status of the zero flag is inspected. If the zero flag is equal to 1, the following signals are sent:

- register r15 is asked to put its value on bus1
- register PC is asked to read its value from bus1
- the ALU is disconnected
- bus0 and bus2 are not connected to any registers

However, when the zero flag was equal to 0, the following signals are sent instead:

- register PC is asked to increment
- the ALU is disconnected
- bus0, bus1, and bus2 are not connected to any registers

When the clock goes to 0, register PC will contain the right value, depending on the state of the zero flag.

Chapter 3

The Moncky-2 processor

3.1 Instruction set

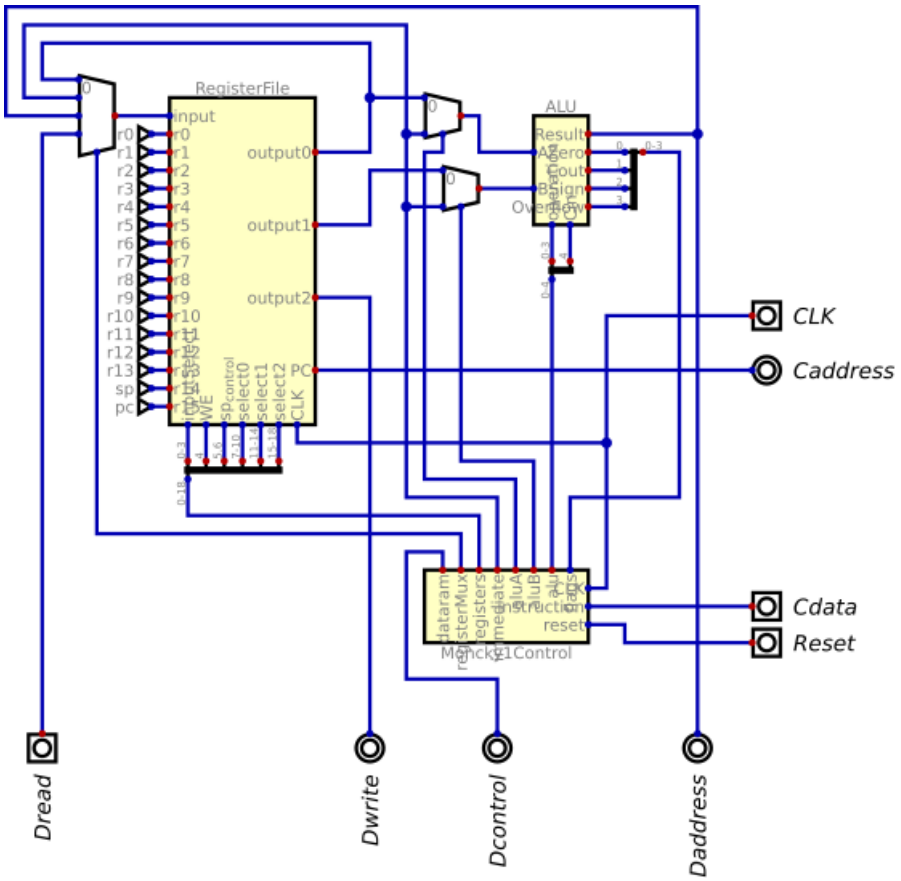
The Moncky-2 processor has exactly the same functionality as the Moncky-1. It only adds two instructions:

- reset (opcode 0010): will put the program counter back to 0. The processor automatically executes this instruction when the external 'Reset' input is set to 0.
- pcto (opcode 1110): will save the program counter to a register. This instruction was originally also available in the Moncky-1 but was removed to make the instruction set as small as possible. This instruction makes procedure calls easier so it was reintroduced.

3.2 Architecture

The architecture of the Moncky-2 processor is very different. The design can be implemented on an FPGA as there are no busses any more. They are replaced by MUXs.

The architecture is shown in the following diagram:



The registers have one input and three outputs. This is much easier to implement than the Moncky-1 registers. The control inputs determine which register is put on which output and which register is connected to the input. The input can come from 4 sources: another register (from output0), an immediate (from the control logic), the ALU, or the data RAM.

Register r15 is used as program counter (PC). It will increase automatically at the falling edge of the clock except when a new value is loaded in it. The jump instruction can therefore also be implemented as a 'nop r15, r...' instruction.

Both the inputs of the ALU can be connected to a register or an immediate. This is not necessary for the Moncky-2 instruction set but it enables other instructions, introduced in the Moncky-3.

The address sent to the data RAM always comes from the ALU. This is not necessary but again allows for future expansion of the instruction set (see Moncky-3).

Although the architecture is much different, the functionality remains the same. The Moncky-2 processor can still execute each instruction in one clock cycle.

Even though it is possible to make an FPGA version of this processor, this has not been done. Instead the architecture was used to create the Moncky-3 processor which has a lot more instructions. There is however a fully working implementation in Digital available:

`https://gitlab.com/big-bat/moncky`

Chapter 4

The Moncky-3 processor

The Moncky-3 processor is an extension of the Moncky-2 processor in many different ways:

- register r14 can be used as a stack pointer. It can increment or decrement while being used
- an interrupt line was added. When interrupts are enabled, the processor will respond to a rising edge on this input by calling a fixed address in memory. The stack is used to be able to return to the original code.
- there are many instructions (currently 32) that enable the code to be much faster. Every instruction still executes in one clock cycle.
- the Moncky-3 processor exists as a simulation in Digital. It has also been implemented on an FPGA using Verilog as an HDL. It forms the heart of the Moncky-3 computer which is explained in chapter **??**.

4.1 Architecture

The architecture is based on that of the Moncky-2 processor. This means that:

- all connections are made using MUX's so it is possible to implement the processor on an FPGA
- the program counter is stored in register r15. This register automatically increments when it is not written to. If the register is read, the output is set to PC+1. In this way the return address can be read immediately when a procedure call is done.
- register r14 can be used as a stack pointer. There are two control lines indicating what should happen to this register:
  - 00 indicates that r14 is used as a normal register
  - 01 increments r14. This is used to 'pop' a value from the stack
  - 10 decrements r14. This is used to 'push' a value on the stack
  - 11 decrements r14. This is used for a hardware interrupt. When the control lines are both equal to 1, the value of r15 will be equal to PC (and not PC+1) so that the processor will return to the current instruction after the interrupt (instead of to the next).

- register 13 is wired to output pins so they can be used to drive external hardware. The pins can currently only be used as digital outputs.

There are also some differences:

- the processor has an 'enable' input. If set to 0, it will hold the processor from executing an instruction even when the clock ticks. This input can also be seen as a 'hold' input which is active low. The enable input is used to control the write enable inputs on all flip flops in the processor. This can for instance be used to synchronise the processor with the memory.
- the processor has input/output instructions. They behave like memory input/output instructions but they cause the 'io_request' output to be set to 1. This allows the use of separate memory for input/output.
- the processor contains interrupt control circuitry. Three flip flops are used to control the interrupts:
  - 'interrupts enabled' indicates if the interrupts are accepted or not. The state of this flip flop can be controlled by the instructions 'ei' (enable interrupts) and 'di' (disable interrupts).
  - 'last interrupt' is used to hold the previous value of the interrupt line. It is used to detect a rising edge.
  - 'interrupt granted' indicates if the requested interrupt is granted. It will only be equal to 1 during one clock cycle. This signal causes the processor to ignore the current instruction and perform a hardware interrupt instruction.

4.2 Instruction set architecture (ISA)

The ISA of the Moncky-3 consists of 4 groups of instructions:

- instructions without operands
- instructions with one operand of 4 bits
- instructions with two operands of 4 bits each
- instructions with three operands of 4 bits each or two operands of 4 and 8 bits respectively

The group is determined by the values of the least significant bits of the instruction as follows:

instruction	group	meaning
`pppp 0000 0000 0000`	0	opcode=pppp, no params
`rrrr pppp p000 0000`	1	opcode=ppppp, 1 param rrrr
`rrrr ssss pppp 0000`	2	opcode=pppp, 2 params rrrr and ssss
`rrrr ssss tttt pppp`	3	opcode=pppp, 3 params rrrr, ssss, and tttt
`rrrr iiii iiii pppp`	3	opcode=pppp, 2 params rrrr and iiiiiiii

Remark that the opcode of group 1 has 5 bits, which overlaps with group 2. This means that opcodes 0000 and 1000 of group 2 are not valid (they belong to group 1).

The same happens for opcode 00000 in group 1. This overlaps with group 0 and is therefore not allowed.

The following instructions are available:

machine language	meaning	example
0000 0000 0000 0000	PC=0 and disable interrupts	reset
0001 0000 0000 0000	PC stops incrementing, interrupts are enabled	halt
0010 0000 0000 0000	enable interrupts	ei
0011 0000 0000 0000	disable interrupts	di
0100 0000 0000 0000	return from interrupt, enable interrupts	reti
0101 0000 0000 0000	software interrupt (call 0x10), disable interrupts	int
0110 0000 0000 0000	hardware interrupt (don't use this)	
rrrr 0000 1000 0000	RAM[sp--] = R[r]	push r3
rrrr 0001 0000 0000	R[r] = RAM[++sp]	pop r4
rrrr 0001 1000 0000	RAM[sp--] = PC; PC = R[r]	call [r5]
rrrr 0010 0000 0000	R[r] = flags (4 bits, zero extended)	sflags r1
rrrr 0010 1000 0000	flags = 4 LSB of R[r] (*)	rflags r2
rrrr 0011 0000 0000	PC = R[r]	jp [r3]
rrrr 10ff f000 0000	R[r] = -flags[f] (0000 or FFFF)	snz r4
rrrr 11ff f000 0000	if (flags[f]) PC = R[r]	jpnc r5
rrrr ssss 0001 0000	R[r] = IO[R[s]]	in r3, (r4)
rrrr ssss 0010 0000	IO[R[s]]=R[r]	out r1, (r10)
rrrr iiii iiii 0001	R[r] = 0000 0000 iiii iiii	li r6, 42
rrrr iiii iiii 0010	R[r] = R[r] | iiii iiii 0000 0000	lih r7, 100
rrrr iiii iiii 0011	R[r] = R[r] + ssss ssss iiii iiii (sign extended) (*)	addi r8, -42
rrrr iiii iiii 0100	R[r] = R[r] & 0000 0000 iiii iiii (*)	andi r9, 0x0F
rrrr iiii iiii 0101	R[r] = R[r] | 0000 0000 iiii iiii (*)	ori r10, 0x80
rrrr iiii iiii 0110	set flags for (R[r] - ssss ssss iiii iiii) (*)	cmpi r1, 32
rrrr iiii iiii 0111	set flags for (ssss ssss iiii iiii - R[r]) (*)	cmpir r2, 32
rrrr ssss AAAA 1000	R[r] = R[r] ALU R[s] (*)	add r4, r5
rrrr ssss AAAA 1001	set flags for (R[r] ALU R[s]) (*)	addf r5, r6
rrrr iiii AAAA 1010	R[r] = R[r] ALU 0000 0000 0000 iiii (*)	shli r7, 3
rrrr iiii AAAA 1011	set flags for (R[r] = R[r] ALU 0000 0000 0000 iiii) (*)	shrif r8, 2
rrrr ssss tttt 1100	R[r] = RAM[R[s] + R[t]]	lda r3, (r4+r1)
rrrr ssss tttt 1101	RAM[R[s] + R[t]] = R[r]	sta r1, (r4+r0)
rrrr iiii tttt 1110	R[r] = RAM[tttt + 0000 0000 0000 iiii]	ldi r3, (r4+7)
rrrr iiii tttt 1111	RAM[R[t] + 0000 0000 0000 iiii] = R[r]	sti r2, (r0+1)

All other possible instructions are currently wired to 'nop' (no operation).

The following notation is used:

- R represents the array of registers
- RAM represents the RAM memory containing 65536 16-bit words
- IO represents the IO RAM memory containing 65536 16-bit words
- "iiii iiii" and "iiii" stand for a value that is embedded into the instruction. This is also called an 'immediate' value as the value is immediately available from the instruction. Depending on the instruction the value is sign-extended or not to become 16 bits.
- "rrrr", "ssss", and "tttt" stand for register numbers (0-15)
- "AAAA" stands for an ALU operation. The following operations are supported:
  - 0000 = NOP (No OPeration: the result is equal to the second input of the ALU)

  - 0001 = OR (a bitwise OR)
  - 0010 = AND (a bitwise AND)
  - 0011 = XOR (a bitwise XOR)
  - 0100 = ADD (an addition)
  - 0101 = SUB (a subtraction)
  - 0110 = SHL (shift left: the first number is shifted to the left. The number of bits to shift is indicated by the second number)
  - 0111 = SHR (shift right)
  - 1000 = ASHR (arithmetic shift right)
  - 1001 = NOT (logic inverse: the result is the 1-complement of the second input)
  - 1010 = NEG (negative: the result is the 2-complement of the second input)
  - 1011 = ADDC (add with carry)
  - 1100 = SUBC (add with the inverse of the second parameter plus carry)
  - 1101 = SHLC (shift left with carry. If more than one bit is shifted, the carry bit is repeated)
  - 1110 = SHRC (shift right with carry)

- "fff" indicates a certain condition. Every time a calculation is done by the ALU, 4 flags are set according to the result: the carry flag, the zero flag, the sign flag, and the overflow flag). These flags can be used to decide to jump to a certain location in the code RAM. The following conditions can be checked:
  - 000 = 'z' = if the zero flag is equal to 1 (the result of the calculation was 0)
  - 001 = 'nz' = if the zero flag is equal to 0 (the result of the calculation was not equal to 0)
  - 010 = 'c' = if the carry flag is equal to 1 (the calculation resulted in 1 extra bit being discarded)
  - 011 = 'nc' = if the carry flag is equal to 0 (the calculation did not result in an extra bit)
  - 100 = 's' = if the sign flag is equal to 1 (the result of the calculation was negative)
  - 101 = 'ns' = if the sign flag is equal to 0 (the result of the calculation was not negative)
  - 110 = 'o' = if the overflow flag is equal to 1 (the calculation resulted in an overflow)
  - 111 = 'no' = if the overflow flag is equal to 0 (the calculation did not result in an overflow)

There are 4 flags which are only set when an arithmetic operation is done (these instructions are marked with a (*) in the table above):

- zero flag: indicates if the result was equal to zero
- carry flag: for additions and subtractions it indicates if an extra carry-bit was generated. This indicates an overflow for unsigned numbers (for subtractions the inverse of the carry flag indicates an overflow!). When doing shift operations with offset 1, the carry flag indicates the value of the bit that was shifted out. If the offset is not equal to 1, the resulting carry flag is meaningless.
- sign flag: indicates if the result was smaller than zero (it just gives the value of the most significant bit of the result). This is only relevant for 2-complement numbers.
- overflow flag: after an addition or subtraction, this flag indicates if a 2-complement overflow has occurred. When doing logical operations, this flag indicates the parity of the result by xor-ing all the bits (i.e. if an even number of bits are 1, the flag is set to 0)

4.3 Assembly language

This section describes all assembly instructions more in detail:

reset this will reset the processor by setting the program counter to 0. It will also disable interrupts so they cannot interfere.

halt this instruction causes the processor to stop executing instructions. The PC will not increment any more. The interrupts are enabled so that it is possible to escape from this state with a hardware interrupt

ei enables the interrupts

di disables the interrupts

reti returns from an interrupt. It is the same as 'pop r15' but this instruction will also enable the interrupts so they don't need to be explicitly enabled.

int causes the processor to perform a software interrupt. It is equivalent to 'call 0x10' (the interrupt handler is hard wired to address 0x10), but it will also disable interrupts automatically

hi hardware interrupt: this is not to be used by a programmer. It behaves like the 'int' instruction but the return address will be set to the current address (since the current instruction will not be executed if a hardware interrupt occurs)

push pushes a register onto the stack. This can be used to save the value of a register or to pass the value to a subroutine

pop pops a register from the stack. This can be used to restore a value of a register that was previously saved on the stack

call this will jump to the address stored in a register. At the same time, the return address is pushed onto the stack

sflags saves the flags to a register. The flags consist of 4 bits (from LSB to MSB): zero, carry, sign, overflow. This instruction can be used in an interrupt handler to preserve the state of the flags

rflags restores the flags from a register. The 4 LSB of the register are copied to the flags.

jp this will jump to the value of the given register. This instruction is equivalent to nop r15, ... but it will not change the status of the flags (the nop instruction is regarded as a calculation and will change the flags)

sz,snz,sc,snc,ss,sns,so,sno these instructions will save the status of a flag to a register. If the selected flag is set, the result will be 0xFFFF (0 otherwise). One can also save the inverse of the flag. For instance, 'sz' will save the status of the zero flag, while 'snz' will save its inverse. These instructions are very useful when conditions need to be combined (with and, or, not, ...). They can also be used to avoid conditional jumps (see chapter **??**).

jpz,jpnz,jpc,jpnc,jps,jpns,jpo,jpno these instructions will jump to the address stored in a register, but only if the selected condition is true

in reads data from input/output memory at the address stored in a register. The data is stored in the given register. For instance, 'in r0, (r1)' will use r1 for the address and will store the result into r0.

out writes data to input/output memory. For instance 'out r0, (r1)' will store the value of r0 at the address given by r1.

li loads a value (an 'immediate') into a register. The value must be positive and can only be up to 8 bits long. It is zero-extended to fit into the register.

lih loads a value into the MSB of a register. The LSB of the register will not be changed. The value must be a positive 8-bit number.

addi this will add a value to a register. The value must be an 8-bit 2-complement number. It is sign-extended before being added.

andi this will do a bitwise AND of a register and a value. The value must be an unsigned 8-bit value. It is zero-extended.

ori this will do a bitwise OR of a register and a value. The value must be an unsigned 8-bit value. It is zero-extended.

cmpi compares a register with a value by subtracting the value from the register. The result is not stored. Only the flags are stored. The value must be an 8-bit 2-complement value. It is sign-extended before being subtracted from the register

cmpir compares a register with a value by subtracting the register from the value. The result is not stored. Only the flags are stored. The value must be an 8-bit 2-complement value. It is sign-extended before subtracting the register.

nop,or,and,xor,add,sub,shl,shr,ashr,not,neg,addc,subc,shlc,shrc these are instructions that do an ALU operation on two registers. The flags are set and the result is stored into the first operand

nopf,orf,andf,xorf,addf,subf,shlf,shrf,ashrf,notf,negf,addcf,subcf,shlcf,shrcf these are instructions that do an ALU operation on two registers. The result is not stored. Only the flags are set

nopi,xori,subi,shli,shri,ashri,noti,negi,addci,subci,shlci,shrci these instructions do an ALU operation on a register and an immediate value. The value must be an unsigned 4-bit number. The result is stored in the given register. The addi, ori, and andi instructions are left out because there exists a better version (described above) with 8-bit values. However, the opcodes for these instructions exist and could be used.

nopif,xorif,subif,shlif,shrif,ashrif,notif,negif,addcif,subcif,shlcif,shrcif these instructions do an ALU operation on a register and an immediate value. The value must be an unsigned 4-bit number. The result is not stored. Only the flags are set.

lda will load a value from the data memory. The address is given by the sum of the values of two registers and the result is stored in another register. This instruction can be useful to access elements in an array.

sta will store a value to data memory. The address is given by the sum of the values of two registers.

ldi will load a value from the data memory. The address is given by the sum of the value of a register and an immediate value. The value must be an unsigned 4-bit number. If the immediate is equal to 0, this corresponds to the 'ld' instruction of the Moncky-1.

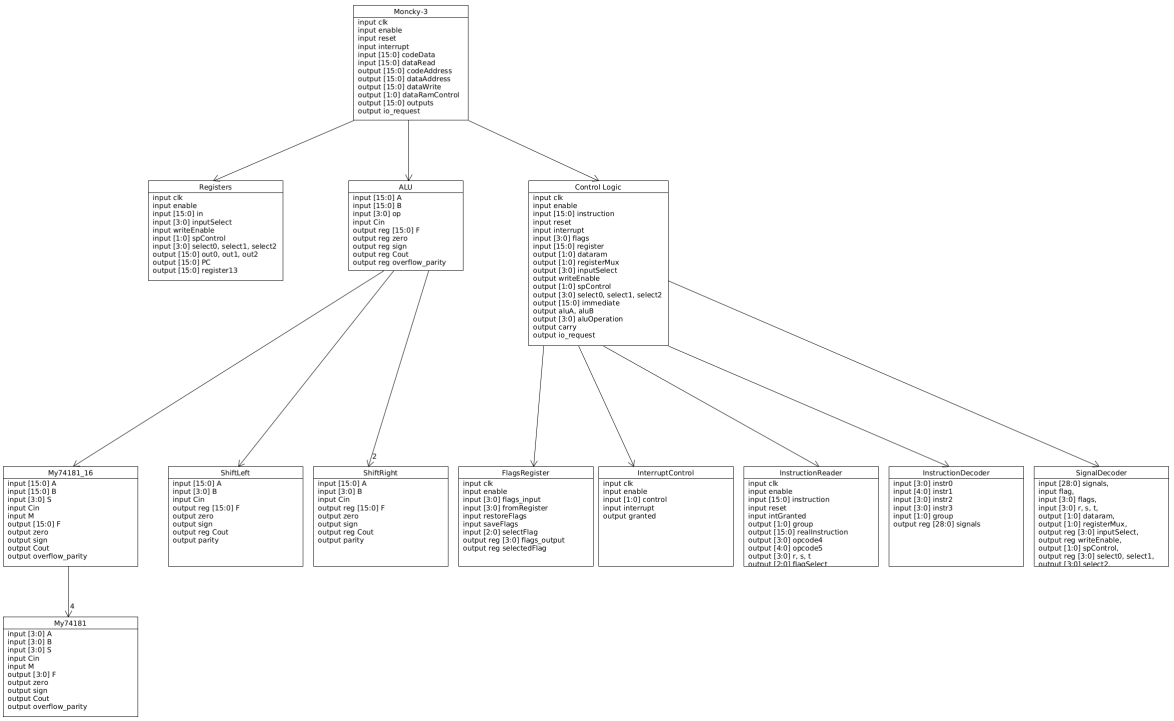
sti will store a value to data memory. The address is given by the sum of the value of a register and an immediate value. The value must be an unsigned 4-bit number. If the immediate is equal to 0, this corresponds to the 'st' instruction of the Moncky-1.

4.4 Implementation in Verilog

The Moncky-3 processor was implemented in Verilog HDL. You can find the code on gitlab:

`https://gitlab.com/big-bat/moncky`

A schematic overview of all the modules can be seen here:



The Moncky-3 processor has the following connections:

- clk: this is the main clock. Each instruction is performed at the falling edge of the clock
- enable: if 0 the clock is disabled. This can be used to synchronise the processor with memory
- reset: active high. If 0 the processor performs the 'reset' instruction
- interrupt: the processor will perform a hardware interrupt when a rising edge on this connection is detected and interrupts are enabled
- codeAddress, codeData: these are used to read instructions from the code memory
- dataAddress, dataRead, dataWrite, dataRamControl: these are used to read and write data from and to the data memory
- outputs: this is connected to register r13. It allows to directly drive external hardware using this register
- io_request: this indicates an input/output request. Used in conjunction with codeAddress, codeData, dataWrite, and dataRamControl

The Moncky processor contains an ALU which is based on the 74181 chip. The circuit was altered so that it uses positive logic for data and control signals. Four of these circuits are then combined into a 16-bit version. The shift operations are implemented separately. MUX's are used to select the right result (74181 or shift circuits).

The control logic consists of different parts:

- FlagsRegister is a 4-bit register containing the values of the flags
- InterruptControl contains the 3 flip flops to control the interrupts
- InstructionReader disassembles the instruction into its different parts
- InstructionDecoder decodes the instruction into control signals that are internally used (in the control logic)
- SignalDecoder will translate the control signals into control signals for the ALU, the registers, and the MUX's