A Poorman's MIPS Architecture

February 14, 2016

Student:
David Veenstra
10094423

Docent:
Prof. dr. Jan van Eijck

Course:
Functional Specification of Algorithms

1 Introduction

Haskell has a treasure chest full of interesting abstraction, often borrowed or inspired by category theory. The monoid, the functor and the monad abstractions are a few examples. Especially the last one, the monad, has been of great importance, as it allowed an otherwise pure language to deal with IO operations in elegant manner.

Besides the more common abstractions mentioned above, Haskell also have a number of more exotic abstractions, such as the lens, comonad, bifunctor, profunctor and arrow. This paper will take a look at the arrow abstraction.

Arrows where originally designed to solve a problem with parser combinators using monads, such as the *parsec* library. The problem is simple. A parser cannot partially discard the input to save memory while parsing, because the parser might fail, where upon an alternative parser might need to be applied on the entire input.

Arrows can be seen as a circuit, and the combinators can be used to build larger circuit existing out of smaller circuits.

One of Haskell advantages is that often an implementation can be given that is idiomatic in the specific domain of the problem. Again the parser combinators are a great example, the resulting implementation often has a strong resemblance to the grammar of what has to be parsed. Given the similarity between circuits and arrows it is to be expected that it should be natural to simulate a computer architecture with the use of arrows.

Another advantages of Haskell, is that composition has a central position.

This paper will evaluate arrows as an abstraction of computation, by implementing a simple computer architecture. In particular, arrows will be evaluated on the ease of use, and how idomatic the implementation is in the

domain of the problem.

This paper is structured as follows. Firstly, an introduction to arrows is given and a few details of the yampa library are highlighted. Subsequently, the PIPS architecture is outlined. Thirdly, the details of the implementation are presented. Afterwards some crude benchmark are given for the simulation. And finally the evaluation of arrows as an abstraction is given.

2 Arrows

$$\begin{aligned} m &= \frac{|(\vec{x_0} - \vec{x_1})\vec{w}|}{||\vec{w}||} \\ &= \frac{|(-1-b)-(1-b)|}{||\vec{w}||} \\ &= \frac{|-2|}{||\vec{w}||} \\ &= \frac{2}{||\vec{w}||} \end{aligned}$$

2.1 Yampa

3 Pips

The poorman's Mips architecture, or Pips in short, is a simple, single cycle architecture, with no fancy features like pipelining or caching. Similar to the Mips architecture, all instructions have the same width. In this case, one instruction can be represented by a 32-bit number. Furthermore, Pips has a registry and memory where it can store 32-bit numbers, and it has logic to perform integer arithmetic and conditional jumps.

The execution of one instruction is completed in one cycle. This process can be split into three different stages. In the first stage the instruction is fetched from the instruction memory and it is decoded. Based on this information, the control unit sends signals to the different components to configure their operations. In the second stage, the needed values are loaded from the registry if needed and the Arithmetic Logic Unit (ALU) can be used to perform integer arithmetic. In the final stage, data can be written back to the registery or data can be stored into or loaded from memory. And it is calculated which instruction has to be executed next.

The instruction set of Pip is subset of the official Mips instruction set. See Table 1 for the different instruction and their syntax.

Figure 1 show a schematic of the architecture. Some simplification were made. First, the instruction that is retrieved from the instruction memory is a 32-bit number; the field that are split from the instruction vary in the number

of bits. Because some components expect input with a certain number of bits, extra hardware is needed for compatibility. However, in the simulation this hardware is left out, as it would be represented by 32-bits number regardless. Second, the memory in MIPS is byte-addressed, the PIPS architecture is word-addressed. And finally, the schematic doesn't show the control unit and the control lines to prevent clutter.

The instruction retrieval and decoding can be seen on the left. The 32-bit number that is retrieved is split in different ways and in different fields, and is send to the control unit, the register, the PC mutex and ALU mutex.

The second stage can be seen in the middle of the schematic. The ALU performs binaire operation. The first argument originates from the register. To determine the second argument, the ALU mutex is used. This can, for example, be the memory address, in a LW or SW instruction. In addition to the result of the binary operator, the ALU also has a second output, which is 1 if the result is 0, and return 0 otherwise. This is useful for the branch instructions.

The third stage can be seen on the right and upper part of the schematic. The control unit determines if data has to written to the register, or to memory. The writeback mutex is used to determine what data has to be written back to the register. The PC mutex determines what the next program counter is, which determines what instruction is fetched in the next cycle.

Syntax	Example	Semantics
ADD ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	ADD $res, $1, $2	$\$res = \$1 + \$2$
SUB ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	SUB $res, $1, $2	$\$res = \$1 - \$2$
AND ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	AND $res, $1, $2	$\$res = \$1 \ \& \ \$2$
OR ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	OR $res, $1, $2	$\$res = \$1 \mid \$2$
XOR ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	XOR $res, $1, $2	$\$res = \$1 \oplus \$2$
SLT ⟨reg⟩, ⟨reg⟩, ⟨reg⟩	SLT $res, $1, $2	$\$res = \begin{cases} 1 & \$1 < \$2, \\ 0 & \text{otherwise} \end{cases}$
SLL ⟨reg⟩, ⟨reg⟩, ⟨imm⟩	SLL $res, $1, 16	$\$res = \$1 \ll 16$
SRL ⟨reg⟩, ⟨reg⟩, ⟨imm⟩	SRL $res, $1, 16	$\$res = \$1 \gg 16$
ADDI ⟨reg⟩, ⟨imm⟩	ADDI $res, $1, 10	$\$res = \$1 - 10$
LUI ⟨reg⟩, ⟨reg⟩, ⟨imm⟩	LUI $res, 10	$\$res = \$1 \mid (10 \ll 16)$
LW ⟨reg⟩, ⟨label⟩, ⟨reg⟩	LW $el, array, $index	$\$el = Mem[\$array + \$index]$
SW ⟨reg⟩, ⟨label⟩, ⟨reg⟩	SW $index, array, $el	$Mem[\$array + \$index] = \$el$
BEQ ⟨reg⟩, ⟨reg⟩, ⟨label⟩	BEQ $1, $2, Loop	jump to Loop if $\$1 = \$2$
BNE ⟨reg⟩, ⟨reg⟩, ⟨label⟩	BNE $1, $2, Loop	jump to Loop if $\$1 \neq \$2$
J ⟨label⟩	J Exit	jump to label Exit
JR ⟨label⟩	JR $3	jump to instruction $3

Table 1: Pips Instruction Set

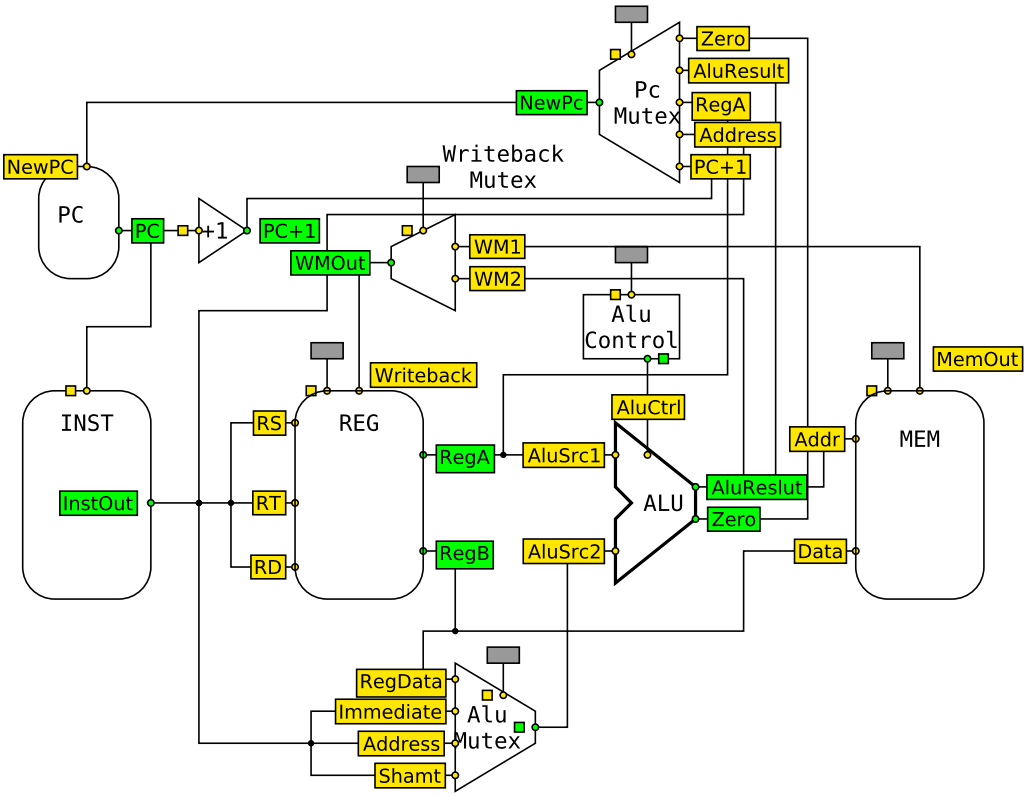

Figure 1: Schematic of the Pips instruction. A green labels indicate an output port. A yellow label indicate an input port. The components that are connect to the gray boxes, are connected to the Control Unit.

4 Implementation

4.1 Benchmark

Experiment	Dataset	
	Origineel	Gereduceerd
random split*	0.755454545455	0.812785388128
cross-validation	0.735454545455	0.748858447489

Table 2: De precisie van de SVM classifier bij de verschillende experimenten. *Gezien dit experiment niet deterministisch is, is het gemiddelde van tien pogingen gebruikt.

5 Evaluation

6 Discussion