

CSE240B Final Project

D.J. Capelis

January 17, 2009

Abstract

The fundamental principles of ISA design have remained the same for decades. The gigahertz wars of the 1990s adversely impacted ISA design. For a period of at least a decade, ISA design has been stagnant. Finally, with the rise of multicore platforms, this is starting to change. We are seeing active research on dataflow architectures and ISAs which have been forgotten for years, if not decades. This makes today an exciting time to be an ISA designer. This project tried to re-capture some of the spirit of the original ISAs and create something rather different, whacky and perhaps a little fun.

Contents

1	Vision	2
1.1	Reverse Code and Data	2
1.2	Modularity	2
1.2.1	Heterogenous Multicore	2
1.3	Cheap Communications Mechanism	3
2	Using Memory	3
2.1	Memory Improvements	3
3	Instructions	4
3.1	Core Instructions	4
3.2	Loops	4
3.3	Conditions	5
3.4	Flow Control	5
3.5	OMI - Operation Module Instructions	5
4	Appendix A: Code Samples	6
4.1	Main AES Loop	6
4.2	Main AES Loop	6
4.3	Alternative Construction With an OMI Which Supports a ShiftRows Instruction	7

1 Vision

We don't have modern ISAs anymore. We simply have old ones. What few new ISAs we do have were created to showcase a specific hardware project. It was the hardware that dictates the ISA instead of the other way around. Given that one of the key issues multicore architectures face is that of making programs which can properly utilize the available die areas we now have, it seems it is time for the ISA to dictate the hardware.

The ISA presented here is not the ISA we need. However, I hope it is a step in a different direction. It is time for a new direction, be it this or another. Incremental improvement of existing ISAs and architectures is not acceptable anymore. It is time to revive the art of ISA design and create one which will properly exploit modern hardware.

1.1 Reverse Code and Data

One of the main differences this ISA was designed to address was how in most modern processors, data flows to the code and not the other way around. So portions of this ISA came to be simply by taking instructions meant specifically for code or data, and making them operate on the other in this ISA.

For instance, load and store instructions from traditional ISAs have operated on data. In this project, we envisioned a world where load and store instructions operated on code. It is because of this that the exec, fork, wait and kill instructions were created.

1.2 Modularity

In my 141L project at UCSD my group ended up with an ISA that contained half of its instruction set dedicated to modules which provided advanced functionality. While our default modules took care of all the functionality needed for the class, it would have been fairly easily to incorporate a mini mips machine into this modular architecture. The power provided by this modularity was palpable.

1.2.1 Heterogenous Multicore

I decided it would be interesting to pair this opportunity for modularity with heterogenous multicore architectures. The idea that different cores would expose completely different sets of instructions has always fascinated me. One could easily imagine a chip with quite a number of cores, only some of which had cryptographic accelerators, or graphics accelerators or floating point units. You certainly wouldn't want a whole new ISA for each of these cores.

It would be best if a program could simply switch portions of its instruction set on the fly. The hardware could work with the operating system to provide the program with the core which interpreted the correct instruction set. In our

implementation, instructions in this range are called operation module instructions (OMI) and the modules which implement instructions in this range are referred to as OMI modules. A program can simply issue an OMI instruction followed by a number to request a certain capability.

1.3 Cheap Communications Mechanism

Also important in multicore architectures is the abundance of cheap intra-process and inter-process communication mechanisms. In this project, we implemented some very lightweight hardware synchronization primitives which allows for a limited form of free/busy bits.

2 Using Memory

Memory is an interesting issue in this particular architecture. Since the idea behind the architecture was to reverse code and data, the memory model is based around the concept of a program counter. The difference is that these are data counters. All memory access is done through registers in this fashion.

2.1 Memory Improvements

This architecture is in need of solid rules for memory ordering based of exec call ordering and the ordering of loop counters.

The following registers are magical:

\$rdaddr This register can be used to read or set the global read address.

\$rladdr This register can be used to read or set the loop read address.

\$wraddr This register can be used to read or set the global write address.

\$wladdr This register can be used to read or set the loop write address.

\$rd This register can be used to read the memory pointed at the global read address.

\$rl This register can be used to read the memory pointed at by the loop read address.

\$wr This register can be used to write to the memory pointed at by the global write address.

\$wl This register can be used to read the memory pointed at by the loop write address.

\$cnt This register can be used inside a loop to read the current iteration of the loop, or immediately outside the loop to see how many times the loop executed. It cannot be written to.

To address memory, simply use any instruction to place an address in one of the four address registers, this will move one of the data counters. Then use any instruction to read from one of the actual data registers and use the results of this information in the program.

3 Instructions

The following is a list of all instructions present in the ISA:

3.1 Core Instructions

These are the instructions that are considered core to the basic functionality of the ISA.

mov src, dst Moves src to dst.

set immed, dst Sets the dst register to the value in the immediate

3.2 Loops

The loop instruction can be formatted in one of three ways:

- loop immediate, (stride), (stride)
- loop reg, (stride), (stride)
- loop condition, (stride), (stride)

In each of these, the stride fields are optional, depending on which loop instruction is used. The first two formats allow for a loop to run a pre-determined number of iterations.

l immed—reg—cond Loop without modifying any of the loop read/write pointers

lr immed—reg—cond, stride Loop and modify the loop read pointer by stride each cycle

lw immed—reg—cond, stride Loop and modify the loop write pointer by stride each cycle

lrw immed—reg—cond, rstride, wstride Loop and modify the loop read pointer by rstride and the loop write pointer by wstride

3.3 Conditions

Condition instructions either are used to determine when a loop ends or are used to execute the next instruction only if the condition is true.

cmp reg, reg—immed Compare a register against another register or an immediate and test if they are the same.

ncmp reg, reg—immed Test if a register against another register or an immediate do not equal

gt reg, reg—immed Test if a register is greater than a register or an immediate.

gte reg, reg—immed Test if a register is greater than or equal to a register or an immediate.

lt reg, reg—immed Test if a register is less than a register or an immediate.

lte reg, reg—immed Test if a register is less than or equal to a register or an immediate.

3.4 Flow Control

These instructions control the execution flow of the program:

exec addr Execute the code at the address pointed to by `addr`

fork addr Schedule the segment of code at the address pointed to by `addr` to be executed, execution continues normally in the current code

wait addr Wait until a write occurs to `addr`

kill addr Kill any current process waiting on `addr`

3.5 OMI - Operation Module Instructions

The follow instructions provide access to various OMI modules and arithmetic instructions:

omi immed Code changes to use OMI instruction set number `immed`. This may require moving the current execution to a different core.

omi0-7 Reserved for the following basic instructions which every OMI module must implement: `add`, `sub`, `sll`, `srl`, `sra`, `and`, `or`, `xor`.

omi7-255 Reserved for the OMI module.

4 Appendix A: Code Samples

4.1 Main AES Loop

This is a representative example of implementing the generic functions of AES. This ends up turning into a pipelined implementation.

```
AES:
    set data, $rdaddr
    set writebase, $5
    add $5, 33, $1
    fork subbytes
    mov $1, $wraddr
    mov $rdaddr, $wr
    add $5, 66, $2
    fork shiftrows
    add $5, 99, $3
    fork mixcolumns
    add $5, 132, $4
    exec addroundkey
```

4.2 Main AES Loop

This is an example of the ShiftRows step of AES as implemented in this language. We use the default OMI module which includes such niceties as a *mod* instruction.

```
ShiftRows:
    wait $2
    mov $2, $wraddr
    mov $rladdr, $wr
    add $2, 32, $6
    ncmp $2, $6
        fork ShiftRows
    add $rladdr, 8, $rladdr
    lr    4, -1
        sub $rladdr, 1, $1
        mod $1, 4, $1
        add $1, $5, $wraddr
        mov $r1, $wr
    add $rladdr, 4, $rladdr
    lr    4, -1
        sub $rladdr, 2, $1
        mod $1, 4, $1
        add $1, $5, $wraddr
        mov $r1, $wr
    add $rladdr, 4, $rladdr
```

```

lr      4, -1
        sub $rladdr, 3, $1
        mod $1, 4, $1
        add $1, $5, $wraddr
        mov $rl, $wr
mov $3, $wraddr
mov $rdaddr, $wr
exec ShiftRows

```

4.3 Alternative Construction With an OMI Which Supports a ShiftRows Instruction

This is an example of the power of the OMI option:

```

ShiftRows :
    wait $2
    mov $2, $wraddr
    mov $rladdr, $wr
    add $2, 32, $6
    ncmp $2, $6
        fork ShiftRows
ShiftRows (OMI Instruction)
mov $3, $wraddr
mov $rdaddr, $wr
exec ShiftRows

```