

# Giving a New ISA a Go

D.J. Capelis  
CSE240B

Final Project Presentation

# New ISA?

- New ISA design is mostly dead
- Parallel architectures deserve new ISAs
- So why not try one?

My hope

Radical Redesign

# The ISA

loop instructions: **l, lr, lw, lrw**

conditions: **cmp, gt, gte, lt, lte**

flow control: **exec, fork, wait, kill**

basic instructions: **set, mov**

operation module instructions: **omi, omi0-  
omi255**

Not so radical?

That doesn't look so bad...

# Did you see...

- Branch instructions?
- Loads?
- Stores?
- Add?

*No. They aren't there.*

At least, not like we're used to...

# Basic Idea

Let's see what happens if we treat code as data and data as code.

*In particular, code should move to where data is, not the other way around*

We now jump to **different data locations**  
instead of different code locations.

Access memory through magical registers,  
have some of them autoincrement at times.

Gee, that sounds like a program counter!

SIMD is easy.

As it turns out, so is MIMD

# I/O

`%rdaddr` - Read pointer  
`%wraddr` - Write pointer  
`%rl` and `%wl` for loops (later)

# I/O: Doing it

Use Magical Registers:

```
mov $rd, $1
```

```
mov $1, $wr
```

What do we do with the old load/store instructions?

Load should now load code.

# Flow Control

Exec – Loosely similar to a jump

Fork – Same thing as a jump except it forks

Store should now store code...

???

# Hardware Sync

Wait – Wait for write to this addr  
Kill – Kills threads waiting on this addr

# Turing Completeness

Condition instructions:

- cmp
- gt
- gte
- lt
- lte

Cool, now that we're turing complete...  
How about some loops?

# Loop instructions

Loop instructions:

- l - loop
- lr – stride implicit read pointer during loop
- lw – stride implicit write pointer during loop exec
- lrw – stride implicit write and read pointers during loop exec

# Loop Format

loop: immed || reg, (stride), (stride)  
loop : instr, (stride), (stride)

# Sample Loop

```
strlen:
```

```
    set str, $r1addr
```

```
    lr cmp $r1, 0
```

```
    ret $cnt <- magical register
```

- Speculation moves from code to data
- Just run the loop with a jump in the implicit read pointer
- We don't add parallelism, we just help expose it
- Possibly running on multiple cores at once, transparently.

Okay, how does math work?

# Modules

- Operation Module Instructions
- Heterogeneous architectures
- App declares: I want a core that supports omi 0 (basic operations support)
- Code migrates to that core.
- Same thing for omi 31 (basic math + cryptographic acceleration support)

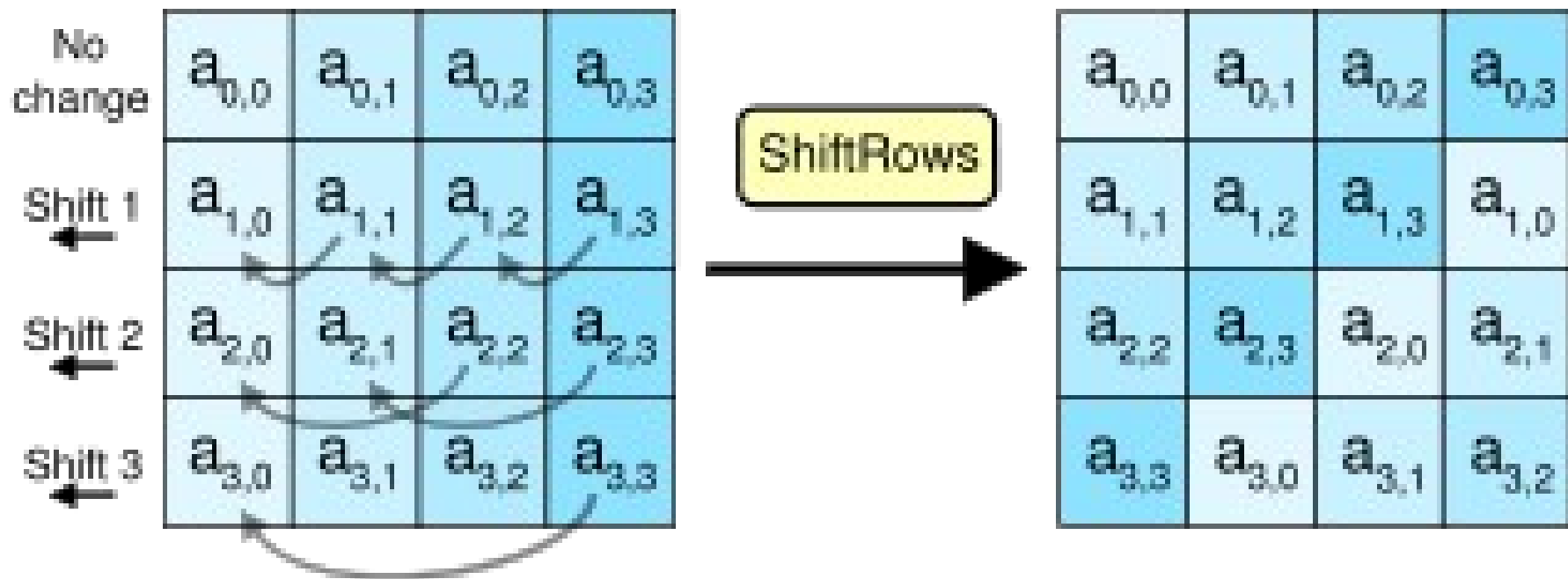
# AES Snippet

AES:

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

# Shift Rows

- Shift things:



# AES Snippet II

ShiftRows:

```
wait $2
```

ExtraShiftRows:

```
add $rladdr, 8, $rladdr
```

```
cmp $2, $rladdr
```

```
fork ExtraShiftRows
```

```
lr 4, -1
```

```
sub $rladdr, 1, $1
```

```
mod $1, 4, $1
```

```
add $1, writebase, $wraddr
```

```
mov $r1, $wr
```

*(two more very similar lr loops)*

```
mov $3, $wraddr
```

```
mov $rdaddr, $wr
```

```
exec ShiftRows
```

# AES Snippet II

ShiftRows:

```
wait $2
```

ExtraShiftRows:

```
add $rladdr, 8, $rladdr
```

```
cmp $2, $rladdr
```

```
fork ExtraShiftRows -> New ones on demand
```

```
lr 4, -1
```

```
sub $rladdr, 1, $1
```

```
mod $1, 4, $1
```

```
add $1, writebase, $wraddr
```

```
mov $r1, $wr
```

*(two more very similar lr loops)*

```
mov $3, $wraddr -> Wakes up mixcolumns
```

```
mov $rdaddr, $wr
```

```
exec ShiftRows
```

# Alternate Implementation

```
ShiftRows:
```

```
    wait $2
```

```
ExtraShiftRows:
```

```
    add $rladdr, 8, $rladdr
```

```
    cmp $2, $rladdr
```

```
        fork ExtraShiftRows
```

```
oid 31
```

```
oid125 (ShiftRows)
```

```
mov $3, $wraddr
```

```
mov $rdaddr, $wr
```

```
exec ShiftRows
```

# Out of Time

Questions?