A SIMD MIPS (In Verilog)

EE685, Fall 2024

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

Parallel Machines

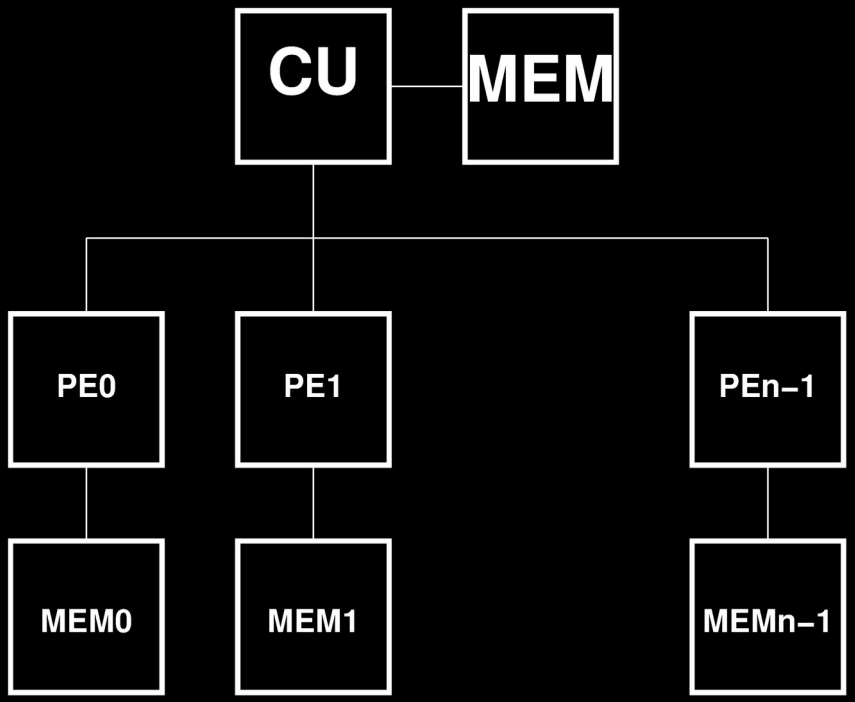
- There are two flavors of large-scale parallelism:
  - MIMD: different program on each PE (multi-core processors, clusters, etc.)
  - SIMD: same instruction on PE's local data (GPUs – graphics processing units, SWAR – SIMD within a register)

Parallel Machines

- There are two flavors of large-scale parallelism:
  - MIMD: different program on each PE (multi-core processors, clusters, etc.)
  - SIMD: same instruction on PE's local data (GPUs – graphics processing units)
- Each MIMD PE runs a sequential program... nothing special in code generation
- SIMD machines are different:
  - If one PE executes some code, all must
  - Can disable a PE that doesn't want to do it

SIMD Concepts

- One Control Unit, many Processing Elements
- MEM contains instructions, ***scalar*** data
- MEM0..MEMn-1 contains only ***parallel*** data



SIMD Code

- There are two flavors of data
  - Singular, Scalar: one value all PEs agree on
  - Plural, Parallel: value local to each PE
- Assignments and expressions work normally, except when mixing singular and plural:
  - Singular values can be copied to plurals
  - Plural values have to be “reduced” to a single value to treat as singular; for example, using operators like `any` or `all`
- Control flow is complicated by enable masking...

if (*expr*) *stat*

- Jump over *stat* if *expr* is false for all PEs; otherwise, do for all the PEs where it's true

```
   PushEn                ;save PE enable state
   {code for expr}
   Test                  ;test on each PE...
   DisableF              ;turn off if false
   Any                   ;any PE still enabled?
   JumpF L               ;any PE must do stat?
   {code for stat}
L: PopEn                 ;restore enable state
```

`if (c < 5) a = b;`

- Masking idea can be used in sequential code to avoid using control flow: if conversion
- The above can be rewritten as:

```
a = ((c < 5) ? b : a);
```

- Bitwise AND with -1 can be used to enable, while AND with 0 disables, thus simply OR:

```
t = -(c < 5);
a = ((t & b) | ((~t) & a));
```

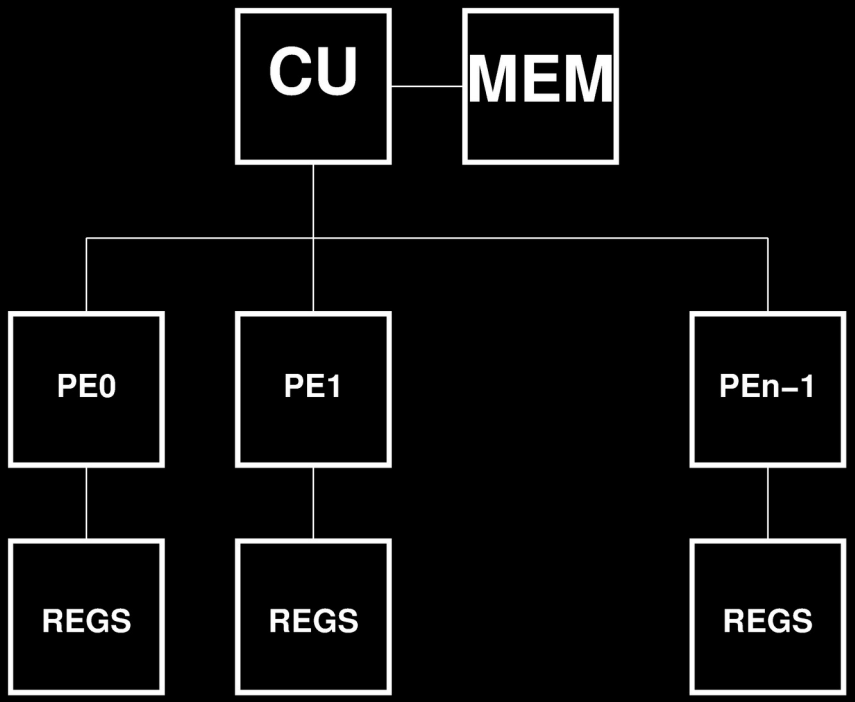
while (*expr*) *stat*

- Keep doing *stat* while *expr* is true for any PE; once off, PE stays off until while ends

```
     PushEn                ;save PE enable state
M:   {code for expr}
     Test                  ;test on each PE...
     DisableF              ;turn myself off if false
     Any                   ;any PE still enabled?
     JumpF L               ;exit if no PE enabled
     {code for stat}
     Jump M
L:   PopEn                 ;restore enable state
```

MIPS-Based SIMD

- CU is a MIPS processor with memory
- PEs are simplified MIPS that only have regs
- SWAR in both + provisions for enable, com



RTYPE Instructions: CU

- We'll use MIPS for the CU instructions:

```
addu  $rd,$rs,$rt      $rd = $rs + $rt
sltu  $rd,$rs,$rt      $rd = $rs < $rt
and   $rd,$rs,$rt      $rd = $rs & $rt
or    $rd,$rs,$rt      $rd = $rs | $rt
xor   $rd,$rs,$rt      $rd = $rs ^ $rt
subu  $rd,$rs,$rt      $rd = $rs - $rt
addu8 $rd,$rs,$rt      SWAR addu, 8-bit
```

- No reason not to have PE versions too...

RTYPE Instructions: PE

- Need new PE versions:

`paddu`	`$rd,$rs,$rt`	$rd = $rs + $rt
`psltu`	`$rd,$rs,$rt`	$rd = $rs < $rt
`pand`	`$rd,$rs,$rt`	$rd = $rs & $rt
`por`	`$rd,$rs,$rt`	$rd = $rs | $rt
`pxor`	`$rd,$rs,$rt`	$rd = $rs ^ $rt
`psubu`	`$rd,$rs,$rt`	$rd = $rs - $rt
`paddu8`	`$rd,$rs,$rt`	SWAR addu, 8-bit

- Not really very different...

Immediates: CU

- Again, just like MIPS:

`addiu $rt,$rs,imm`	$rt = $rs + imm
`sltiu $rt,$rs,imm`	$rt = $rs < imm
`andi $rt,$rs,imm`	$rt = $rs & imm
`ori $rt,$rs,imm`	$rt = $rs | imm
`xori $rt,$rs,imm`	$rt = $rs ^ imm
`lui $rt,imm`	$rt = imm << 16

Immediates: PE

- Easy, but same immediate for all PEs:

```
paddiu $rt,$rs,imm    $rt = $rs + imm
psltiu $rt,$rs,imm    $rt = $rs < imm
pandi  $rt,$rs,imm    $rt = $rs & imm
pori   $rt,$rs,imm    $rt = $rs | imm
pxori  $rt,$rs,imm    $rt = $rs ^ imm
plui   $rt,imm        $rt = imm << 16
```

Load From Memory: CU

- There's only one memory interface, for CU; `imm + $rs` should be a CU computation:

  `lw $rt,imm($rs)`

  $rt = memory[imm + $rs]

- Result goes in CU `$rt`

Store To Memory: CU

- A lot like load...

```
sw $rt,imm($rs)
memory[imm + $rs] = $rt
```

Control Flow: CU

- Only the CU implements control flow:

  `beq $rs,$rt,lab`

  if ($rs == $rt) pc = (pc + 4) + (offset * 4)

  where offset = (lab – (pc + 4)) / 4

- Of course, offset is really imm… and we shift by 2 rather than multiply by 4

Enable Logic: PE

- Only the PE implements enable logic:

`offeq $rs,$rt`	**Turn off…**
`pushen`	**Push enable state**
`popen`	**Pop enable state**

- We can use the activity counter hack in the PEs – just need one register for this: `$ra`

Communication

- Really nothing like this in MIPS:

```
gor    $rt,$rs          CU $rt = OR(PE $rs)
bcast  $rt,$rs          PE $rt = CU $rs
net    $rd,$rs,$rt      $rd = PE[$rs].$rt
```

- Note that net is owner stores (if enabled)
- The `net` operation is really just a MUX

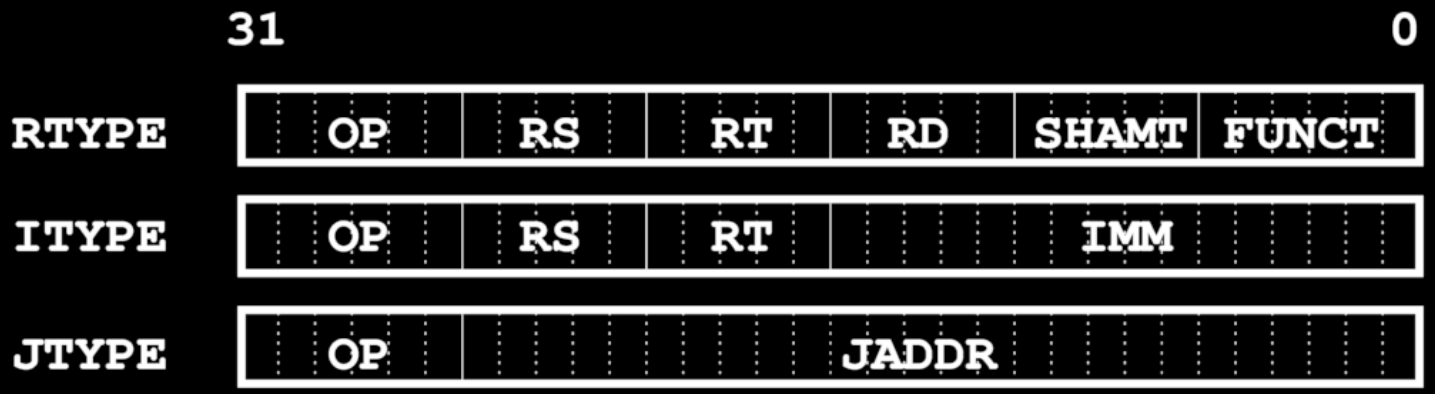
PE Registers

- Some PE registers are special:

`$zero`	`$0`	constant 0 (read only)
	`$1-$27`	general use
`$gp`	`$28`	`$live` (read only)
`$sp`	`$29`	`$iproc` (read only)
`$fp`	`$30`	`$nproc` (read only)
`$ra`	`$31`	`$ac` activity counter

- `$live` is ($ac == 0), 1 if PE is enabled
- `$ac` is number of times disabled

MIPS Instruction Fields




```
// Fields
`define OP    [31:26] // opcode field
`define RS    [25:21] // rs field
`define RT    [20:16] // rt field
`define RD    [15:11] // rd field
`define IMM   [15:0]  // immediate/offset field
`define SHAMT [10:6]  // shift amount
`define FUNCT [5:0]   // function code (opcode extension)
`define JADDR [25:0]  // jump address field
```

Instruction Set Encoding

- We'll use a very simple encoding in which all SIMD instructions have the 16 bit on in the OP
  - OP==16 means PE register instruction
  - PE immediates have OP|16
  - Special PE operations also use OP==16
- I will allow you to change encoding… as long as you also change the AIK assembler spec.

A Bit About AIK

- See `http://aggregate.org/EE480/assembler.html`
- Some things a tad non-standard for MIPS
  - `[]` is used in load/store, not ()
  - `;` is a comment, not #
- Memory is declared as byte addressed, so code is output as bytes in VMEM format

```
.segment .text 8 0x10000 0 .VMEM
.segment .data 8 0x10000 0 .VMEM
```

The Usual Suspects

- The MIPS instructions already implemented:

```
.Reg $rd,$rs,$rt := 0:6 rs:5 rt:5 rd:5 0:5 .this:6
.alias .Reg 32 addu8 addu 35 subu and or xor 43 sltu
.Imm $rt,$rs,imm := .this:6 rs:5 rt:5 imm:16
.alias .Imm 9 addiu 11 sltiu andi ori xori
beq $rs,$rt,lab := 4:6 rs:5 rt:5 ((lab-(.+4))/4):16
.LdSt $rt,imm[$rs] := .this:6 rs:5 rt:5 imm:16
.alias .LdSt 35 lw 43 sw
lui $rt,imm := 15:6 0:5 rt:5 imm:16
```

- Note that **addu8** is in there...

The Usual Suspects in PEs

- PE versions of the MIPS instructions already implemented:

```
.PReg $rd,$rs,$rt := 16:6 rs:5 rt:5 rd:5 0:5 .this:6
.alias .PReg 32 paddu8 paddu net psubu pand por pxor 43 psltu
.PImm $rt,$rs,imm := .this:6 rs:5 rt:5 imm:16
.alias .PImm 25 paddiu 27 psltiu pandi pori pxori
plui $rt,imm := 31:6 0:5 rt:5 imm:16
```

- Note that **paddu8** is in here too…
  and yes, **net** too...

The Unusual Suspects in PEs

- The really strange ones:

```
.PRsRt $rs,$rt := 16:6 rs:5 rt:5 0:5 0:5 .this:6
.alias .PRsRt 0 offeq gor bcast
pushen := 16:6 rs:5 rt:5 rd:5 0:5 4:6
popen := 16:6 rs:5 rt:5 rd:5 0:5 5:6
```

- We already handled coding of **net**

The Built-In Register Names

- Not much to this:

```
.const { zero 2 v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
         s0 s1 s2 s3 s4 s5 s6 s7 t8 t9 28 gp sp fp ra
         28 live iproc nproc ac }
```

- This means `$iproc` works like `$29`

The Complete AIK Spec

```
.Reg $rd,$rs,$rt := 0:6 rs:5 rt:5 rd:5 0:5 .this:6
.alias .Reg 32 addu8 addu 35 subu and or xor 43 sltu
.Imm $rt,$rs,imm := .this:6 rs:5 rt:5 imm:16
.alias .Imm 9 addiu 11 sltiu andi ori xori
beq $rs,$rt,lab := 4:6 rs:5 rt:5 ((lab-(.+4))/4):16
.LdSt $rt,imm[$rs] := .this:6 rs:5 rt:5 imm:16
.alias .LdSt 35 lw 43 sw
lui $rt,imm := 15:6 0:5 rt:5 imm:16
.PReg $rd,$rs,$rt := 16:6 rs:5 rt:5 rd:5 0:5 .this:6
.alias .PReg 32 paddu8 paddu net psubu pand por pxor 43 psltu
.PImm $rt,$rs,imm := .this:6 rs:5 rt:5 imm:16
.alias .PImm 25 paddiu 27 psltiu pandi pori pxori
plui $rt,imm := 31:6 0:5 rt:5 imm:16
.PRsRt $rs,$rt := 16:6 rs:5 rt:5 0:5 0:5 .this:6
.alias .PRsRt 0 offeq gor bcast
pushen := 16:6 rs:5 rt:5 rd:5 0:5 4:6
popen := 16:6 rs:5 rt:5 rd:5 0:5 5:6
.const { zero 2 v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
         s0 s1 s2 s3 s4 s5 s6 s7 t8 t9 28 gp sp fp ra
         28 live iproc nproc ac }
.segment .text 8 0x10000 0 .VMEM
.segment .data 8 0x10000 0 .VMEM
```

The Complete AIK Spec

- Is here:

  `http://aggregate.org/EE685/simdmips.html`

- Actually, there are two there…
  - One as given here
  - A second with one line per instruction