

A SIMD MIPS (In Verilog)

EE685, Fall 2026

Hank Dietz

<http://aggregate.org/hankd/>

Parallel Machines

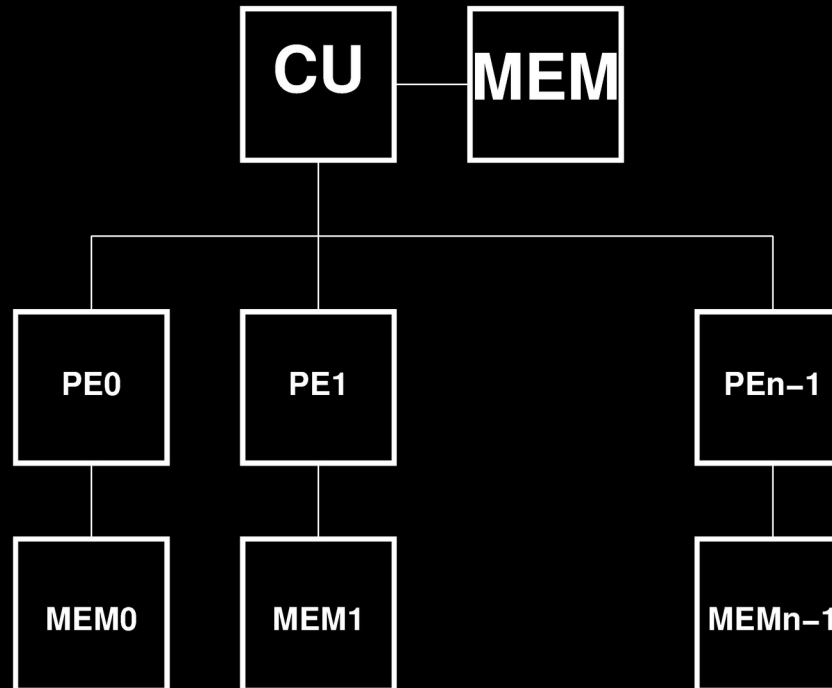
- There are two flavors of large-scale parallelism:
 - **MIMD**: different program on each PE
(multi-core processors, clusters, etc.)
 - **SIMD**: same instruction on PE's local data
(**GPUs** – graphics processing units,
SWAR – SIMD within a register)

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
<i>{code for expr}</i>	
Test	;test on each PE...
DisableF	;turn off if false
Any	;any PE still enabled?
JumpF L	;any PE must do stat?
<i>{code for stat}</i>	
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));
```

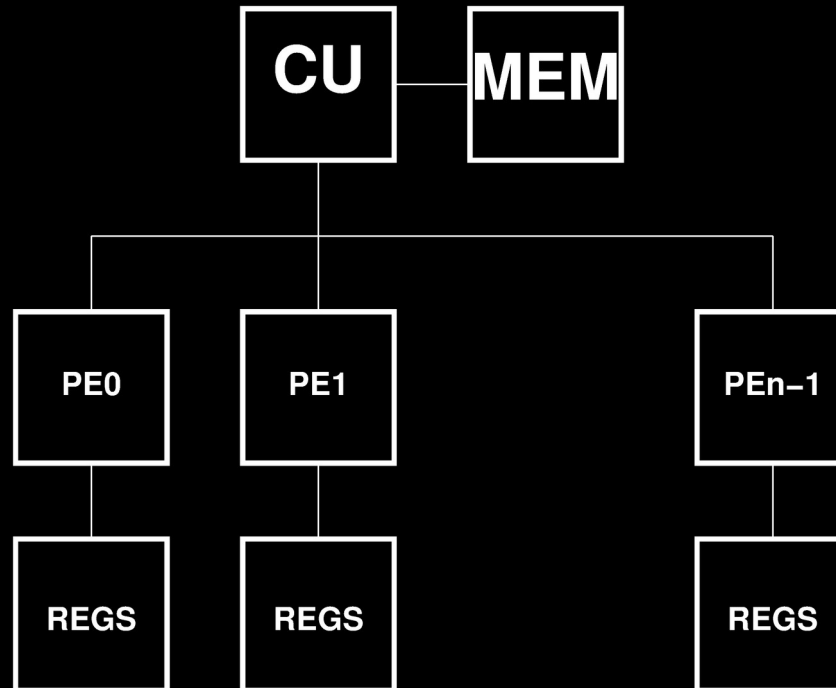
while (*expr*) *stat*

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

```
M:    PushEn                ;save PE enable state
      {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
- Provisions for enable, communication



RTYPE Instructions: CU

- We'll use MIPS for the CU instructions:

<code>addu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs + \$rt$
<code>sltu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs < \$rt$
<code>and</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \& \$rt$
<code>or</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \$rt$
<code>xor</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \wedge \$rt$
<code>subu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs - \$rt$

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

RTYPE Instructions: PE

- Need new PE versions:

<code>paddu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs + \$rt$
<code>psltu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs < \$rt$
<code>pand</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \& \$rt$
<code>por</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \$rt$
<code>pxor</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs \wedge \$rt$
<code>psubu</code>	<code>\$rd, \$rs, \$rt</code>	$\$rd = \$rs - \$rt$

- 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`

$\$rt = \$rs \wedge imm$

`lui $rt, imm`

$\$rt = imm \ll 16$

Immediates: PE

- Easy, but same immediate for all PEs:

<code>paddiu</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs + imm$
<code>psltiu</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs < imm$
<code>pandi</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs \& imm$
<code>pori</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs imm$
<code>pxori</code>	<code>\$rt, \$rs, imm</code>	$\$rt = \$rs \wedge imm$
<code>plui</code>	<code>\$rt, imm</code>	$\$rt = imm \ll 16$

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

- Each PE has a dedicated 32-bit DL – disable level – register that *counts number of nested times disabled* and is initialized to 0

Disable Level Counter

- What instructions do to DL:

<code>offeq \$rs, \$rt</code>	$DL = (DL ? DL : (rs == rt))$
<code>pushen</code>	$DL += (DL != 0)$
<code>popen</code>	$DL -= (DL != 0)$

- A PE is enabled *iff* $DL == 0$
- The DL is the “Activity Counter” in

<https://aggregate.org/GPUCOURSE/keryell93activity.pdf>

Communication

- Really nothing like this in MIPS:

<code>gor</code>	<code>\$rt, \$rs</code>	<code>CU \$rt = OR(PE \$rs)</code>
<code>bcast</code>	<code>\$rt, \$rs</code>	<code>PE \$rt = CU \$rs</code>
<code>net</code>	<code>\$rd, \$rs, \$rt</code>	<code>PE \$rd = PE[\$rs].\$rt</code>

- The `gor` operation is used to read PE results; note that `gor $rt, $live` is SIMD ANY
- The `net` operation is really just a MUX

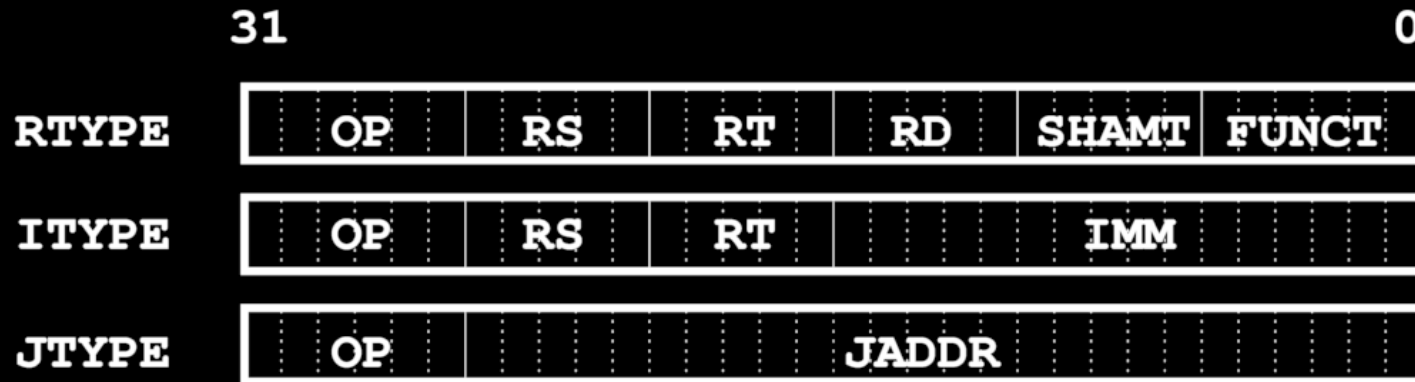
PE Registers

- Some PE registers are special:

<code>\$zero</code>	<code>\$0</code>	constant 0 (read only)
	<code>\$1–\$27</code>	general use
<code>\$gp</code>	<code>\$28</code>	<code>\$d1</code> (read only)
<code>\$sp</code>	<code>\$29</code>	<code>\$iproc</code> (read only)
<code>\$fp</code>	<code>\$30</code>	<code>\$nproc</code> (read only)
<code>\$ra</code>	<code>\$31</code>	<code>\$live</code> (read only)

- `$d1` is DL, the number of levels disabled
- `$live` is (DL == 0), 1 if PE is enabled

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 ALK 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 33 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
```

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 33 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 **net** is in there...

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 dl iproc nproc live }
```

- 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 33 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 33 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 dl iproc nproc live }
.segment .text 8 0x10000 0 .VMEM
.segment .data 8 0x10000 0 .VMEM
```

The Complete AIK Spec

- Is here:

<https://aggregate.org/EE685/F26simdmins.html>

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

Your Project

- Each team has 3-4 members
 - All will not work on every part, but each must be “aware” of all portions of the project
 - Only one submission, but peer evaluations...
- What you will submit: a tar containing
 - Implementor's notes
 - Verilog implementation of CU and 8 PEs
 - ALK specification
 - Any test code, etc.