15-410
"...*RISCy Business*..."

Project 4

Joshua Wise
April 1, 2010

Synchronization

- SIGBOVIK today!
  - 16:00, Rashid Auditorium
    - (That's GHC 4401 to you)
      - (I think)
- Experience a real live academic conference!

Project 4

- Early announcement this semester
  - *Start designing early!*
- Wait until you have a solid kernel to begin implementing
  - As always, a solid P3 is much more important than a shaky P3 and P4

Project 4: MIPS Port

- Qualitatively different from previous P4s
- Porting an existing Pebbles kernel to MIPS
  - *your* Pebbles kernel!
- Goals: learn about...
  - ...designing for portability
  - ...another architecture
  - ...”wacky” non-x86 machines

Outline

- MIPS, and the RISC way of life
  - Motivations for RISC
  - Implementation specifics...
    - ...and implementation artifacts
    - Extending RISC machines
- Concurrency on MIPS
- VM on MIPS
- Wrap-up

Visit to 1983

- **(or: A Quick Note from 18-447)**
- Performance: a big problem with CPUs
- CISC CPUs, like Intel 80286, slow
  - Many cycles needed per instruction
  - Decoding instructions *difficult*
- What if each instruction could be made simpler?
  - More homogeneous?
  - More *orthogonal?*

Visit to 1983, Part 2

- MIPS project started in 1989
  - John Hennessy, Stanford University
- Experimental technologies for higher performance
- Improve *decode* speed
  - Try to reduce instructions down to one of three forms
- Improve *system* speed
  - Get system frequency up

Peace through Superior Frequency

- If we can simplify things...
  - ...can we get the system down to one clock per instruction?

Peace through Superior Frequency

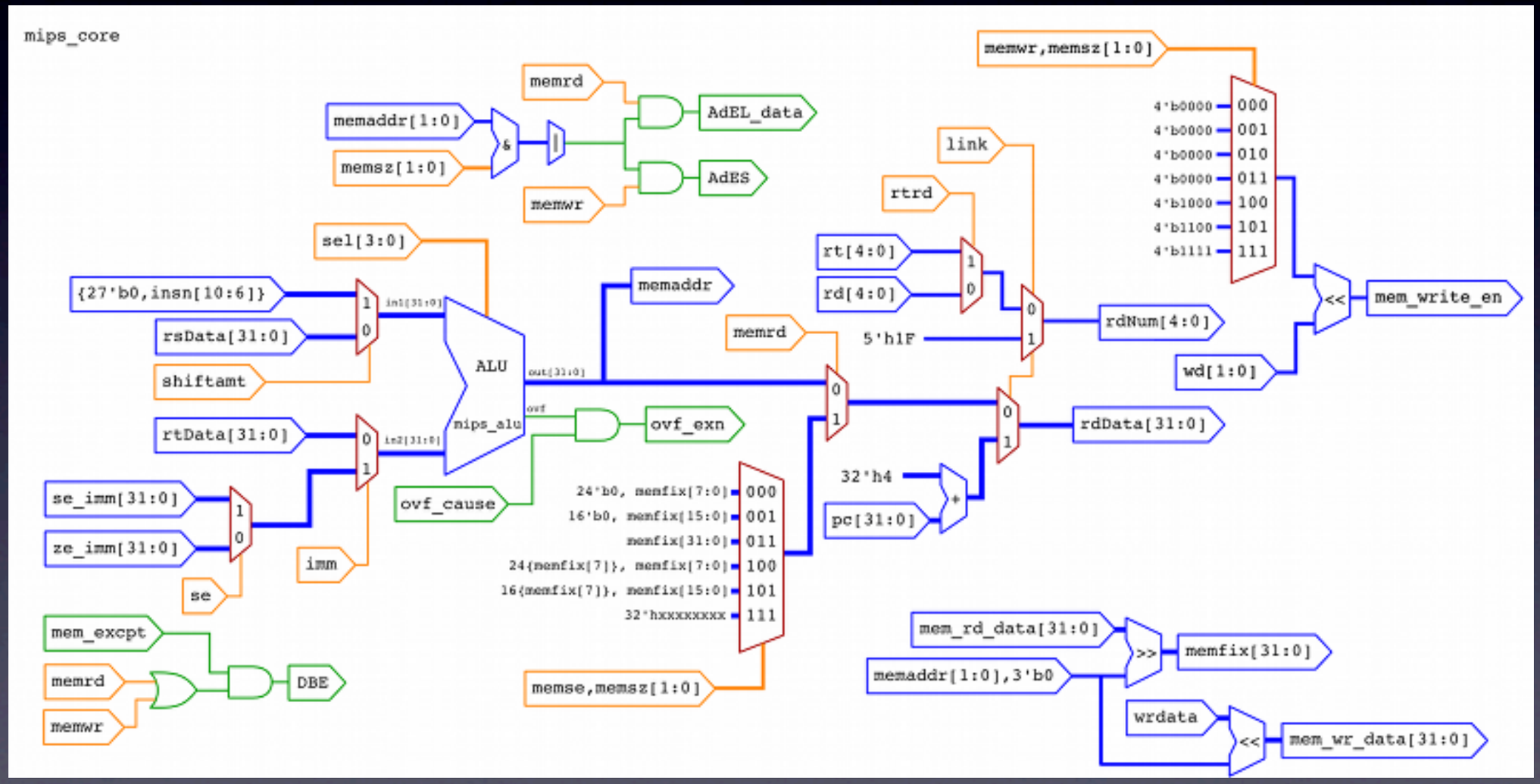


Image Credit: myself, Christopher Lu, Jacob Potter

Actually Superior Frequency

- We got it done in one clock...
  - ...but that is *one slow clock*.
  - *Critical path*: longest path needed for data from one clock to next
- Pipelining
  - What if we *average* one clock per instruction?
  - Each clock, some *part* of an instruction completed
  - Many instructions *in flight*
  - Every clock, one instruction comes out the end

Actually Superior Frequency

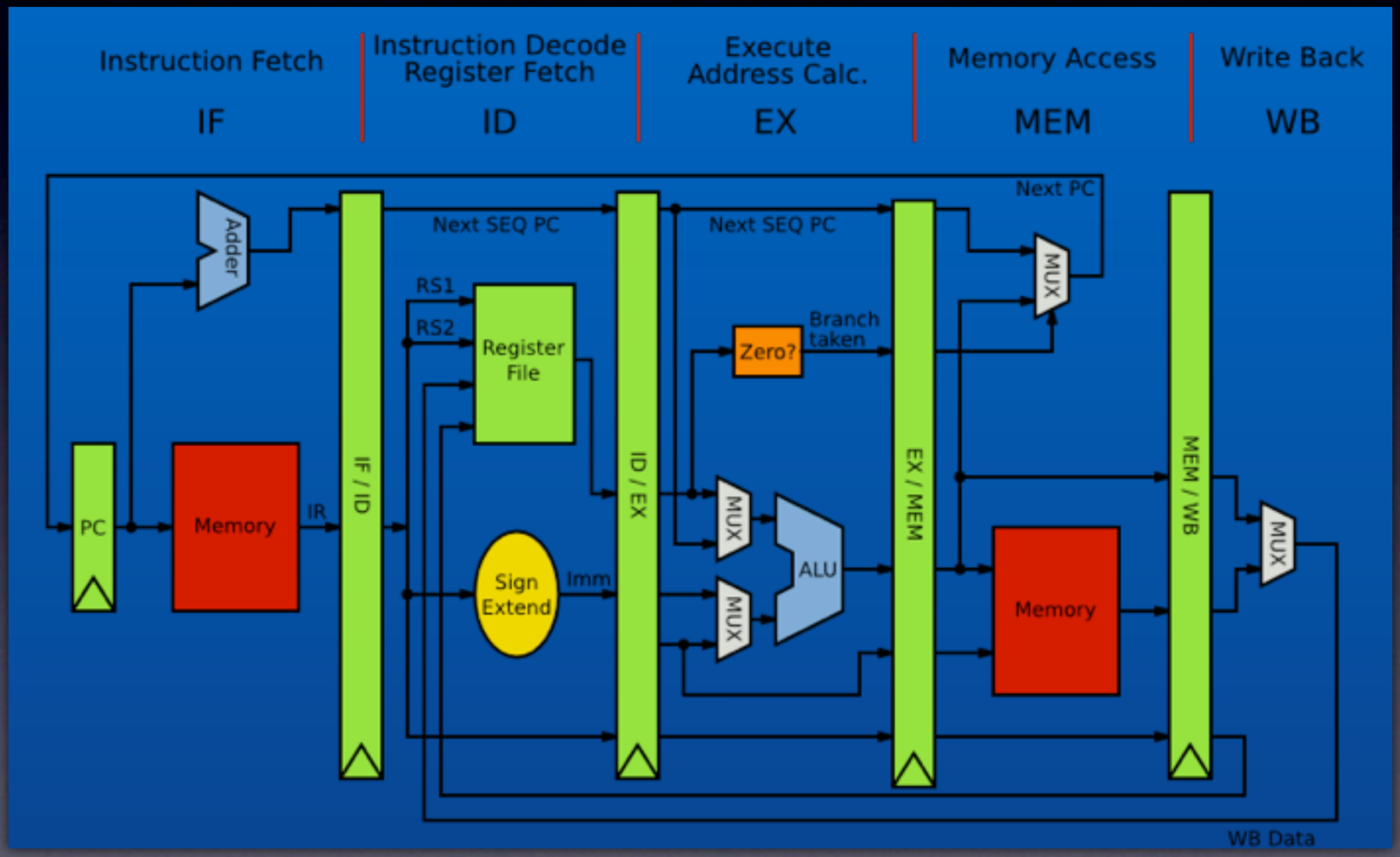


Image Credit: Wikipedia user Inductiveload

Actually Superior Frequency

Image Credit: Wikipedia user Poil

MIPS Specifics

- `</ece>`
  - Want to know more? Consider 18-447
    - (My *other* class this semester!)
- All instructions are four bytes
- 32 registers
  - ...almost -- register `$0` is always zero
- Program counter is not a register!
- "Load-store" machine
  - Cannot directly manipulate memory words
- Other than that, *it's just a computer*

A Look at MIPS Instructions

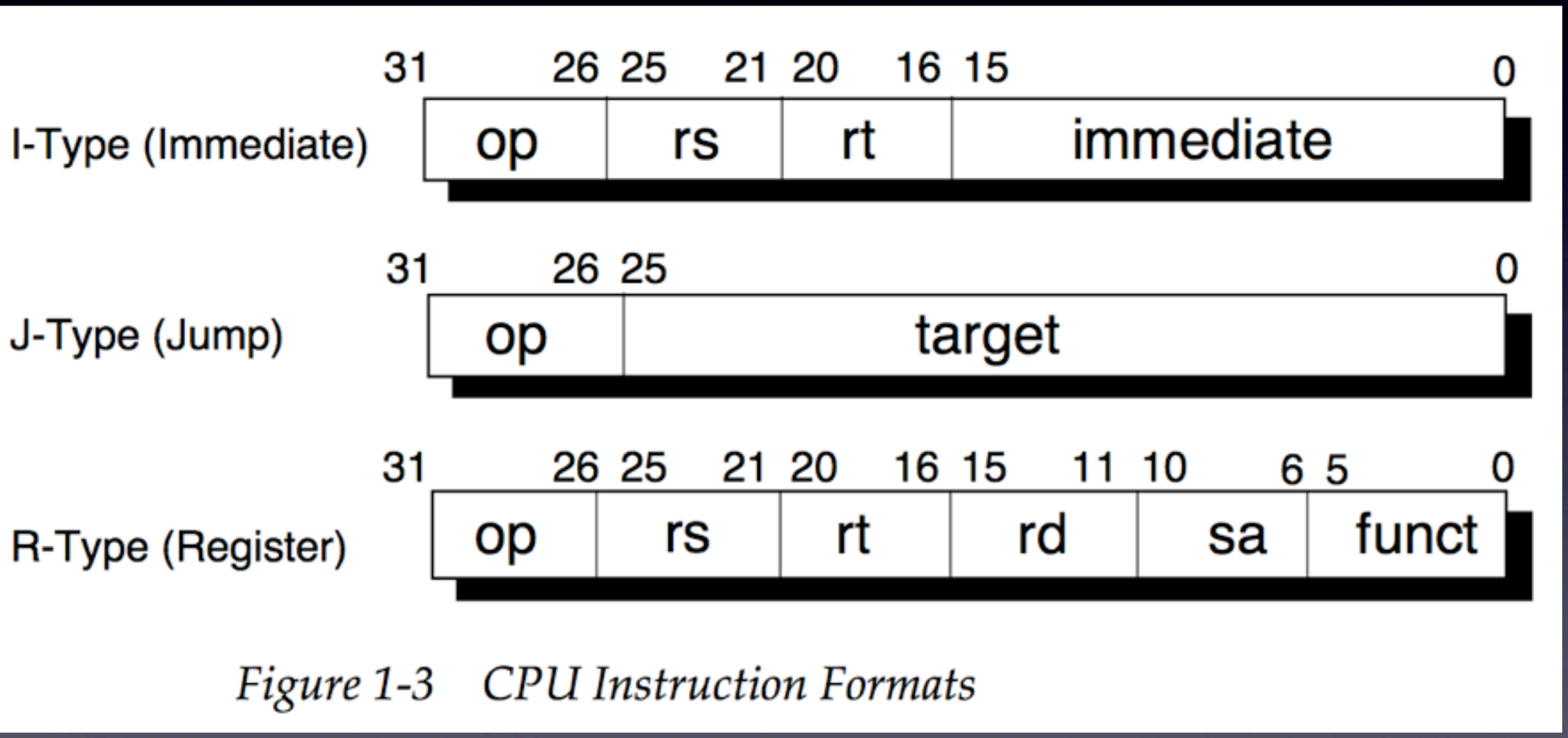


Figure 1-3 CPU Instruction Formats

Image Credit: MIPS R4000 Microprocessor User's Manual, Joe Heinrich

A Look at MIPS Instructions

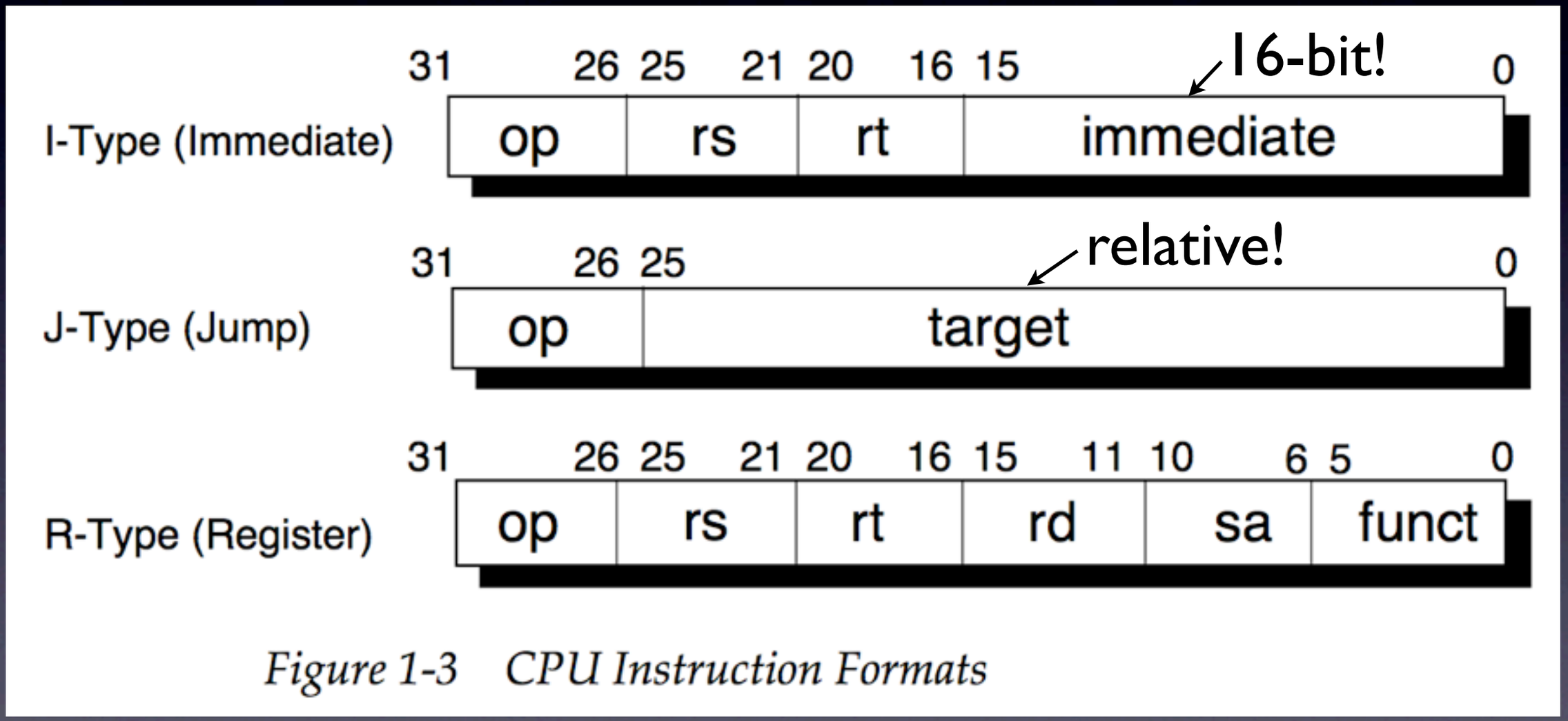
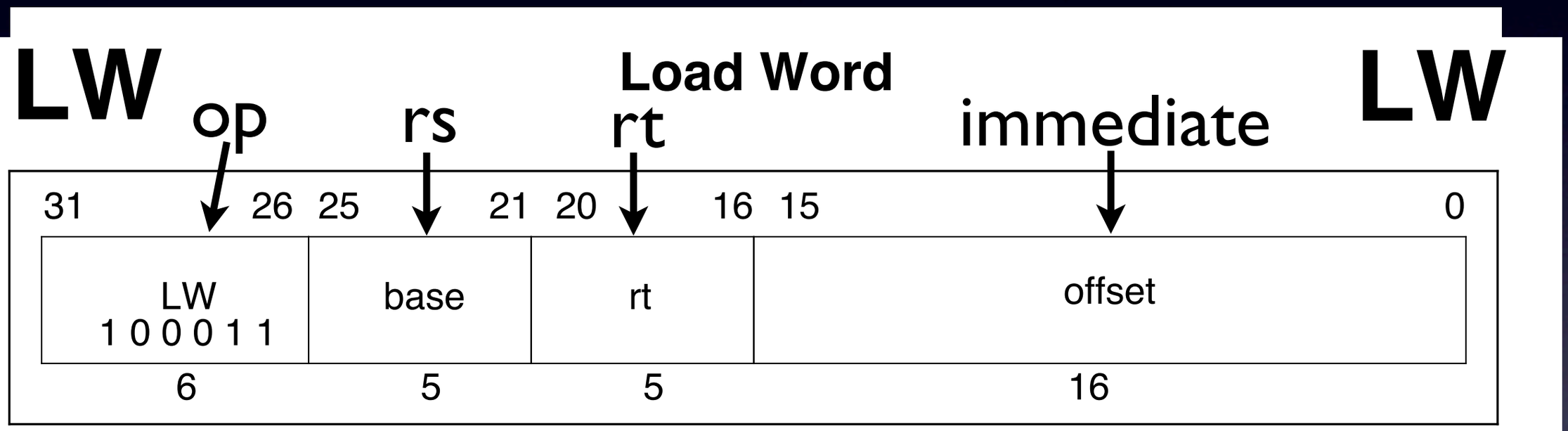


Figure 1-3 CPU Instruction Formats

Image Credit: MIPS R4000 Microprocessor User's Manual, Joe Heinrich

A Look at MIPS Instructions



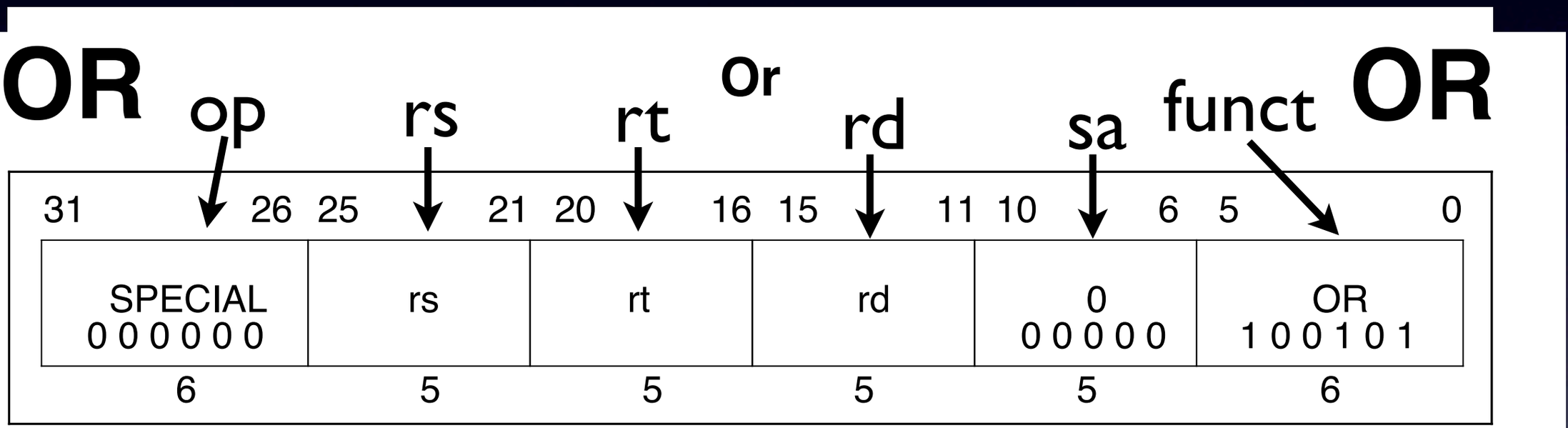
Format:

LW rt, offset(base)

I-type!

Image Credit: MIPS R4000 Microprocessor User's Manual, Joe Heinrich

A Look at MIPS Instructions



Format:

OR rd, rs, rt

R-type!

Image Credit: MIPS R4000 Microprocessor User's Manual, Joe Heinrich

What's interesting about that?

- *They all look the same!*
  - There are only **three** types.

- What's the instruction for a 32-bit immediate?
  - Where do the bits go?
  - ```
    lui $v0, 0x1234          // v0 <- 0x12340000
    addi $v0, $v0, 0x5678    // v0 <- v0 + 0x5678
                             // (v0 = 0x12345678)
    ```
    ```

A Look at Some Code

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

A Look at Some Code

x86

```
// src, dest
swap:
  pushl %ebp
  movl %esp, %ebp
  pushl %ebx

  movl 12(%ebp), %ecx
  movl 8(%ebp), %edx
  movl (%ecx), %eax
  movl (%edx), %ebx
  movl %eax, (%edx)
  movl %ebx, (%ecx)

  movl -4(%ebp), %ebx
  movl %ebp, %esp
  popl %ebp
  ret
```

MIPS

```
// dest, src
swap:
  addiu $sp, $sp, -4
  sw $fp, 0($sp)
  move $fp, $sp

  // arguments passed
  // in registers
  lw $t0, 0($a0)
  lw $t1, 0($a1)
  sw $t0, 0($a1)
  sw $t1, 0($a0)

  move $sp, $fp
  lw $fp, 0($sp)
  jr $ra
  addiu $sp, $sp, 4
```

MIPS Calling Convention

- Since you have so many registers...
  - why not use them all?
- These slides are *not authoritative*
  - A few of the more interesting highlights
  - Real reference in handout some time next week
- Many arguments are in registers, not stacked
  - `$a0` through `$a7`

MIPS Calling Convention

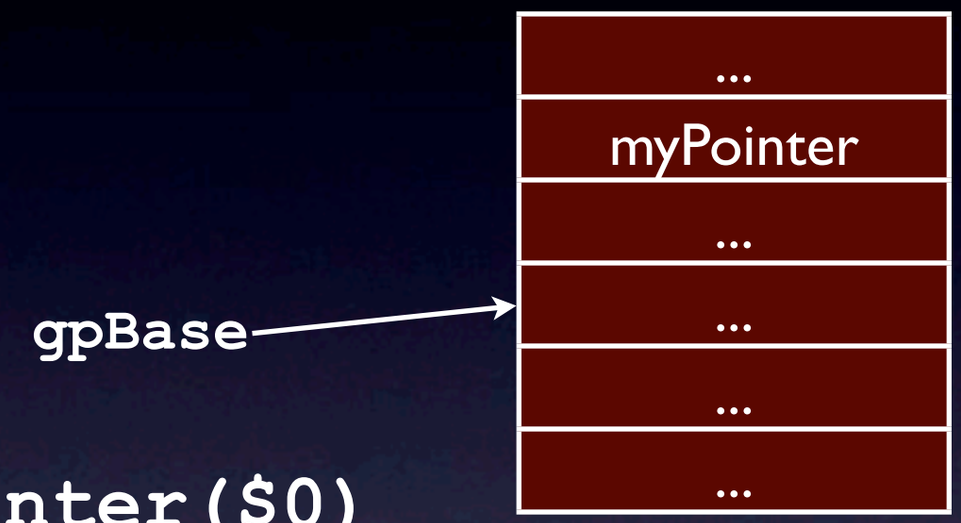
- One register reserved for the assembler -- **`$at`**
  - MIPS instructions small; assembler provides macroinstructions!
  - i.e., **`bgt $rs, $rt, label`**
    - becomes:

```
slt $at, $rs, $rt
bne $at, $zero, label
```

- Two registers reserved for the *kernel*
  - Have to compute an address to save registers to!
  - **`$k0`** and **`$k1`** -- may change *while a user mode program is running*

MIPS Calling Convention

- One "*global pointer*"
  - Offset accesses for global data
  - "Partial pointer"
    - Impossible: `lw $rt, myPointer($0)`
    - Replacement: `lw $rt, gpBase-myPointer($gp)`
    - Preserved throughout lifetime of program
- Other more mundane bits (caller-save, callee-save)
  - See handout, or reference materials on web



Coprocessors

- If instructions are simple...
  - ...then how do we do hard things?
- There *is no* "`CPUID`" instruction
- `CPUID` lives on "coprocessor zero"
- *Don't be fooled by the name!*
  - Coprocessors need not do any actual processing
- Other coprocessors include:
  - FPU
  - Multiply/divide (on some MIPSen)
- Access with special instructions

Magic Coprocessor Zero

- One coprocessor is more magical than the rest
- Present on *every single MIPS*
- Contains controls for:
  - Cache control
  - System identification ("CPUID")
  - Exception/interrupt control
  - Timers, memory control, ...
  - *and virtual memory!*
- VM, PIC, PIT, and northbridge rolled into one

set_cr3?

x86

```
// src, dest
set_cr3:
movl 4(%esp), %eax

movl %eax, %cr3

ret
```

MIPS

```
// dest, src
set_context:
// argument already
// in $v0

jr $ra
mtc0 $v0, C0_CONTEXT
```

C0_CONTEXT is an *optimization*:
"Where you should look" on a fault

Concurrency on Earth

- We know how to do atomic operations on x86
- **`lock xchg`**, discussed in Synch #2
- Implements *one and only one* concurrency mechanism

- Write and read characteristics defined
- ```
  atomic int32 xchg(int32 *mem, int32 new) {
    register int old;
    old = *mem;
    *mem = new;
    return old;
  }
  ```
  ```

Earth Technology is Behind

- What about something else?
  - xchg... conditionally?
  - Atomic addition?
  - Add *and get old value*?
  - *Atomically two's complement memory???*
- CISC solution

Earth Technology is Behind

- What about something else?
  - xchg... conditionally?
  - Atomic addition?
  - Add *and get old value*?
  - *Atomically two's complement memory???*
- CISC solution: new instruction for each
- **`CMPXCHG`, `CMPXCHG8B`, `LOCK ADD`, `LOCK XADD`, `LOCK NOT`**
  - (**`LOCK`**? What's that?)

The Problem with x86 is...

- Adding more instructions is not very RISC-like
- Adds complexity to decode stage
- *Reduces* flexibility of system
- Atomic operations do not fit in pipeline!
  - Two memory operations
  - Waiting is **not** an option

A RISC Solution

- Can we factor out a *common element* from concurrent operations?
  - Do something...

A RISC Solution

- Can we factor out a *common element* from concurrent operations?
- Do something...
- *...only if nobody got in our way.*

A RISC Solution

- Load linked (`LL`)
  - Load memory, and set the link flag
- Link flag
  - *Cleared* when somebody else writes to the same address (cache shootdown, or context switch)
- Store conditional (`SC`)
  - Store memory *only if the link flag is set*
  - Informs programmer if SC failed

Rewriting `mutex_lock`

- LL/SC can be used to implement *any* of those x86 instructions
- Spin-wait in MIPS assembly

```
.1:
    ll $t0, lock_avail($gp)
    bz $t0, .1                  // branch if zero
    addi $t0, $zero, 0          // i.e., $t0 = 0
    sc $t0, lock_avail($gp)     // stores 1 if successful
    bz $t0, .1                  // have to retry if 0
    jr $ra                      // like 'ret'
```

- No special operation needed for unlock!

VM on Earth

- On x86, action centered around page table
- One instruction could cause:
  - ...

VM on Earth

- On x86, action centered around page table
- One instruction could cause:
  - TLB lookup, and miss
  - PDBR (%cr3) read
  - Page directory address calculation
  - Page directory dword read
  - Page table address calculation
  - Page table dword read
  - TLB eviction decision, evict, and store
  - TLB lookup *again*
  - Memory read for actual instruction
- That is a **LOT** of logic!
  - ... and just to handle *one* instruction!

Earthly Ties

- This is in keeping with the CISC way
  - Big complicated task? *Let the hardware deal with it!*
- There is only *one way* to organize pages
  - Want extra bits? *Too bad!*
- Logic required to deal with this is immense
- "Locked in" to one page table format
  - Reserved bits, and PAE/PGE
- "Encapsulate your indefensible code"!

VM, MIPS-Stylin'

- Once again, what is the *common element* to determine if an access is valid?
  - Our friend we love to hate...

VM, MIPS-Stylin'

- Once again, what is the *common element* to determine if an access is valid?
  - Our friend we love to hate...
  - *the TLB!*
- MIPS abstracts virtual memory *only to the TLB level*
- Kernel memory "locked" in the upper half of the address space
  - **`kseg`**
- Let's trace the same case again

Address space

Martian TLB Misses

- TLB lookup for address misses
- Fix needed, *CPU takes exception!*
  - CPU vectors to exception handler
  - EH reads faulting address from cop0
  - *Exception handler* picks "random" entry to evict
    - **`TLBWR`**
  - *Exception handler* reads internal data structures to "fix" fault
  - *Exception handler* traps to kernel if needed
  - *Exception handler* refills TLB
  - **`ERET`** back to retry instruction
- TLB lookup hits; life proceeds

Sample Exception Handler

- The hardware does help
  - If you set up C0_CONTEXT, it will fill in where to read from
- Best case:

```
mfc0 k1, C0_CONTEXT
lw   k0, 0(k1)
lw   k1, 8(k1)
mtc0 k0, C0_ENTRYLO0
mtc0 k1, C0_ENTRYLO1
ssnop                    // Otherwise, the nop
ssnop                    // would get parallelized.
ssnop                    // I'm not joking.
tlbwr
eret
```

TLB Exception Notes

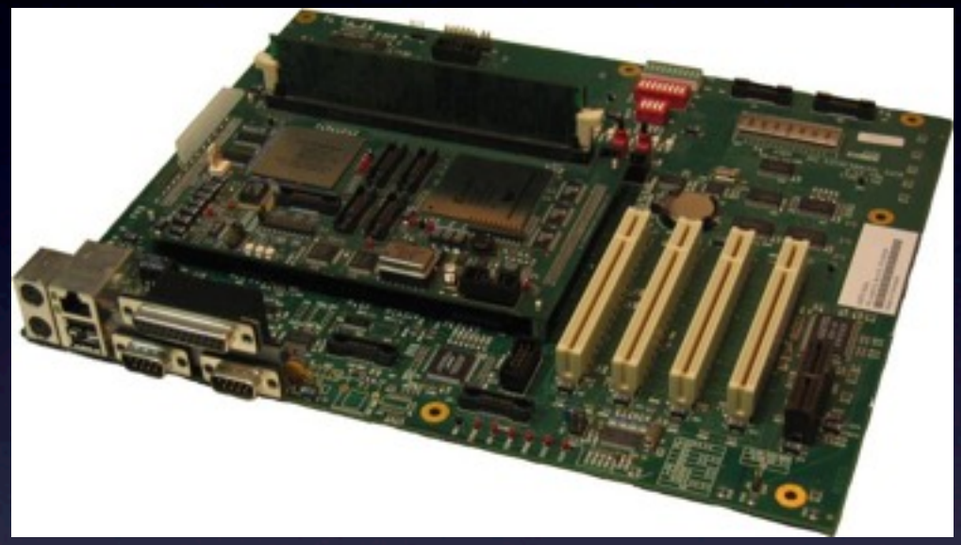
- TLB exception must be *fast*!
  - On MIPS R4K (ours), TLB exceptions usually ~25 instructions
  - Still competitive with x86 TLB miss!
- TLB exception generally follows the same procedure as x86 hardware would
- Exn handler usually not portable
- Looking for hints writing it?
  - `linux/arch/mips/mm/tlbex.c` -- runtime generated
  - `build_r4000_tlb_refill_handler`
  - Yours probably won't be runtime-generated

Exceptions on MIPS

- Exceptions on MIPS also a whole lot less complicated
- MIPS
  - No IDTR, no IDT
  - Exception comes along (TLBL exception!)
  - Processor *immediately vectors* to *fixed* location
- Exceptions *faster*
  - Important if you have a software TLB!
- Exception reason indicated by cop0 **`CAUSE`** register
- We will provide infrastructure to help set these up

Mechanics

- Simics doesn't support our MIPS platform very well
- *Two* methods for you to run your code
  - MIPS Malta
  - Generously donated to Carnegie Mellon when SiCortex went under
  - Debugging using EJTAG probe
  - Instructions soon



SiCortex

Mechanics

- *Two* methods for you to run your code
  - `tmips`: developed by Thomas Tuttle for 18-447
    - Make sure to thank him if you see him!
  - Software simulator
  - Bored before P3 ends?
    - http://www.github.com/ttuttle/tmips

Mechanics

- *Thr*ee methods for you to run your code?
  - *This very MIPS processor* is being implemented by 18-447 students right now!
- Discussed today: MIPS is very simple
  - Simple enough for an *intro comp arch class* to implement almost in its entirety
- 18-447 MIPS is missing a *few* CP0 features...
  - ...but could be trivially extended to support Pebbles
- Excited? Go take 18-447 :-)

Wrap-up

- We talked about:
  - Motivations for RISC
  - Implementation specifics for MIPS
  - Concurrency on MIPS
  - VM on MIPS
  - Exceptions
- P4 handout to be released on day mentioned on 15-410 web site
- Don't spend too much time worrying about MIPS until you have a *solid* kernel!
- Some of these ideas may be on the final!