Two Ways to Program SMIPS

- **Assembly**
  - .S files
  - Register level control of processor
  - Direct translation into machine code (vmh files) by assembler

- **C**
  - .c files
  - Higher level control of processor
  - Compiled by smips-gcc
SMIPS Assembly Files
baseline.S

start:
  mfc0 $28, $10
  li $30, 0
  nop
    ...100 nop’s in total...
  mfc0 $29, $10
  subu $29, $29, $28
  mtc0 $29, $18
  li $29, 10
  mtc0 $29, $19
  mtc0 $30, $21

end:
  beq $0, $0, end

tag for branch instruction

tags
assembly instructions
SMIPS Registers Overview

32 GPR Registers - $0$ to $31$
- $0$ always has the value 0
- Application Binary Interface (ABI) specifies how registers and stack should be used
  - Compiled C programs will typically follow the ABI

32 COP Registers - $0$ to $31$
- Only a few are actually used in our processor
- $10$ – Number of clock cycles passed (R)
- $11$ – Number of instructions executed (R)
- $18$ – Write integer to console (W)
- $19$ – Write char to console (W)
- $21$ – Write finish code (W)
# SMIPS GPR Registers

According to ABI

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Always 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Temporary register for assembler to use</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2 – 3</td>
<td>Method call return values</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4 – 7</td>
<td>Method call arguments</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8 – 15</td>
<td>Temporary register (not preserved during method call)</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16 – 23</td>
<td>Saved register (preserved during method calls)</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24 – 25</td>
<td>Temporary register (not preserved during method call)</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26 – 27</td>
<td>Kernel registers (OS only)</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address</td>
</tr>
</tbody>
</table>
SMIPS Method Calls

Caller

- Caller saves any registers that may get written over by method call
  - $a0 - $a3 – Argument registers
  - $v0, $v1 – Return registers
  - $t0 - $t9 – Temporary registers

- Caller sets argument register(s) $a0-$a3

- Caller jumps to function using jal
  - After call, method will eventually return to instruction after jal

- Get return value(s) from $v0, $v1

- Restore caller-saved registers
SMIPS Method Calls

Method

- Get called
- Move stack pointer to reserve more space on the stack
- Save return address $(ra)$ and saved registers ($s0$-$s7$) to the stack
- Do method including any necessary method calls
- Restore the return address ($ra$) and saved registers ($s0$-$s7$) from the stack
- Move stack pointer to release space on the stack
SMIPS Assembly Instructions

ALU Instructions

- **aluop** $1, $2, $3
  - $1 is the destination
  - $1 ← $2 (aluop) $3

- **aluopi** $1, $2, x
  - x is an immediate value
    - sign-extended for addi, slti, sltiu
    - zero-extended for andi, ori, xori, lui
  - $1 ← $2 (aluop) x

- **shiftop** $1, $2, **shamt**
  - shamt is the shift amount
  - $1 ← $2 (shiftop) x
  - shiftop is shift left logical (sll), shift right logical (srl),
    or shift right arithmetic (sra)
ADDU vs ADD

* Our processor only supports ADDU and ADDIU, not ADD or ADDI
  - ADD and ADDI should cause errors

* Is this a problem?
  - No, ADD and ADDU should give the same output bits regardless of the interpretation of the input bits (signed vs unsigned)

* Why are there different ADD and ADDU instructions then?
  - ADD and ADDI generate exceptions on overflow
  - No one writes programs that use those exceptions anyways...

* But there definitely is a difference between ADDIU and ADDI, right?
  - No, ADDIU still uses a sign-extended immediate value!
SMIPS Assembly Instructions

Memory Instructions

- LW $1, offset($2)
  - $1 <- M[$2 + offset]
  - offset is a signed immediate value
- SW $1, offset($2)
  - M[$2 + offset] <- $1
  - offset is a signed immediate value

There are many unsupported memory instructions in our processor

- Smaller Accesses: LB, LH, LBU, LHU, SB, SH
- Atomic Accesses: LL, SC
  - We will implement these two for the final project
SMIPS Assembly Instructions

Control Flow

- **J address**
- **JAL address**
  - *Address* can be a tag found in the assembly program
  - JAL saves the return address (PC+4) to $ra ($31)
- **JR $1**
  - Jumps to instruction in $1, typically $ra
- **B<op> $1, $2, offset**
  - Jump to PC + 4 + (offset << 2) if $1 <op> $2
  - Example:
    - beq $1, $2, -1 is an infinite loop if $1 == $2
  - Offset can also be a tag found in the assembly program
SMIPS Assembly Instructions

Mnemonics

- **li $1, x**
  - Loads register $1 with sign extended immediate value x
  - Alias for `addiu $1, $0, x`

- **b offset**
  - Always branches to offset
  - Alias for `beq $0, $0, offset`
Writing an Assembly Program

- Add start tag to first instruction
  - This lets the assembler know where the program starts
- Write interesting assembly
  - Include `mtco $??, $18/$19` to print reg $??
  - Include `mtco $??, $21` at end
    - $?? is the register which contains the return code
      - 0 for success, !0 for failure.
- Include infinite loop after final `mtc0`
  - `end: j end`
- Put program in `programs/src/assembly`
- Build program by running `make` in programs
Example Assembly Code

**Assembly if statement:**

```
beq $7, $8, abc
addiu $7, $7, 1
abc: ...
```

**C if statement:**

```
if( $7 != $8 ) {
    $7++;
}
```
Example Assembly Code

Assembly loop:
li $8, 10
begin: addiu $8, $8, -1
bne $8, $0, begin

C loop:
i = 10;
do {
i--;
} while(i != 0);
Assembly Overview

A great way to build low level tests!

- You have control over every instruction and every register
- You can reproduce any processor state with little effort
  - At least for our current pipeline complexity...

A great way to introduce new errors into your testing procedure

- Assembly programming is not easy
C Programs

We have a compiler to turn C programs into SMIPS programs

You can create larger tests and performance benchmarks with ease
C Programs
What’s missing

- smips-gcc sometimes produces unsupported instructions
  - Using types smaller than int (such as char) causes unsupported loads and stores to be implemented
  - Mul and div instructions are unsupported so using * and / causes problems
- No standard libraries
  - Can’t use malloc, printf, etc.
C Programs
What we have

Start code
- Jumps to main and sends return value to COP

Print library
- Can print chars, ints, and strings

Cop library
- Can read number of instructions and things like that.
C Programs

We are going to talk about details in a later tutorial (when we talk about multicore programming)

- If you want to do it on your own, start with an existing example and modify it
- Also add the necessary lines to the makefile
Searchable FIFO
interface SFifo#(numeric type n, type dt, type st);
    method Bool notFull;
    method Action enq(dt x);
    method Bool notEmpty;
    method dt first;
    method Action deq;
    method Action clear;
    Bool search(st x);
endinterface
Searchable FIFO
Internal States

Standard FIFO states:

\[
\begin{align*}
\text{Reg}(\text{Bit}(\text{TLog}(n)))\; & \; \text{enqP} \leftarrow \text{mkReg}(0); \\
\text{Reg}(\text{Bit}(\text{TLog}(n)))\; & \; \text{deqP} \leftarrow \text{mkReg}(0); \\
\text{Reg}(\text{Bool})\; & \; \text{full} \leftarrow \text{mkReg}(\text{False}); \\
\text{Reg}(\text{Bool})\; & \; \text{empty} \leftarrow \text{mkReg}(\text{Empty});
\end{align*}
\]

Need any more?
Searchable FIFO

Method Calls

- `{notFull, enq}`
  - R: full, enqP, deqP
  - W: full, empty, enqP, data
- `{notEmpty, deq, first}`
  - R: empty, enqP, deqP, data
  - W: full, empty, deqP
- `search`
  - R: (empty or full), enqP, deqP, data
- `clear`
  - W: empty, full, enq, deqP
Searchable FIFO

Potential Conflicts

\{notFull, enq\}
- R: full, enqP, deqP
- W: full, empty, enqP, data

\{notEmpty, deq, first\}
- R: empty, enqP, deqP, data
- W: full, empty, deqP

search
- R: (empty or full), enqP, deqP, data

clear
- W: empty, full, enq, deqP

Search is read-only \(\rightarrow\) it can always come first
Clear is write-only \(\rightarrow\) it can always come last
Searchable FIFO
Implementation 1

Implementation:
- `mkCFFifo` with a search method

Schedule:
- `search < {notFull, enq, notEmpty, deq, first} < clear`
- `{notFull, enq} CF {notEmpty, deq, first}`
module mkSFifo1(SFifo#(n, t, t)) provisos(Eq#(t));
  // mkCFFifo implementation

method Bool search(t x);
  Bool found = False;
  for (Integer i = 0; i < valueOf(n); i = i+1) begin
    Bool validEntry = full[0] ||
      (enqP[0]>deqP[0] && i>=deqP[0] && i<enqP[0]) ||
      (enqP[0]<deqP[0] && (i>=deqP[0] || i<enqP[0]));
    if (validEntry && (data[i] == x)) found = True;
  end
  return found;
endmethod
endmodule
module mkSFifo1 (function Bool isFound (dt x, st y), SFifo#(n, dt, st) ifc);

  // mkCFFifo implementation

method Bool search(st x);

  Bool found = False;

  for (Integer i = 0; i < valueOf(n); i = i+1) begin
    Bool validEntry = full[0] ||
      (enqP[0]>deqP[0] && i>=deqP[0] && i<enqP[0]) ||
      (enqP[0]<deqP[0] && (i>=deqP[0] || i<enqP[0]));

    if (validEntry && isFound(data[i], x)) found = True;
  end

  return found;
endmethod

endmodule
Scoreboard

When using a SFifo for a scoreboard, the following functions are used together:

- \{search, notFull, enq\}
- \{notEmpty, deq\}

Are enq and deq still commutative like in the CFFifo case?

- No! Search has to be able to be done with enq, and search is not commutative with deq
Two SFifo Implementations for a Scoreboard

Implementation 1:
- \{\text{search, notFull, enq}\} < \{\text{deq, notEmpty}\}
- “Conflict Free” Scoreboard
  - Can be implemented with previously shown SFifo

Implementation 2:
- \{\text{deq, notEmpty}\} < \{\text{search, notFull, enq}\}
- “Pipeline” Scoreboard
  - Design is straightforward using technique from Lab 4