EETimes article 08/07/2014
RISC-V: An open standard for SoCs. The case for an open ISA (Krste, Patterson, UC Berkeley).
While the likely first beachhead for RISC-V is the IoT, our ambitious goal is grander: Just as Linux has become the standard OS for most computing devices, we envision RISC-V becoming the standard ISA for all computing devices.

ANGEL

• http://riscv.org/angel/
Review

• Memory is byte-addressable, but `lw` and `sw` access one word at a time.

• A pointer (used by `lw` and `sw`) is just a memory address, so we can add to it or subtract from it (using offset).

• A Decision allows us to decide what to execute at run-time rather than compile-time.

• C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.

• MIPS Decision making instructions are the conditional branches: `beq` and `bne`.

• New Instructions: `lw`, `sw`, `beq`, `bne`, `j`
In addition to word data transfers (lw, sw), MIPS has byte data transfers:

- load byte: lb
- store byte: sb

same format as lw, sw

E.g., lb $s0, 3($s1)
- contents of memory location with address = sum of “3” + contents of register s1 is copied to the low byte position of register s0.
What do with other 24 bits in the 32 bit register?
- `lb`: sign extends to fill upper 24 bits

Currently, when loading bytes, the upper 24 bits are filled with the sign-extended value of the byte. However, this is not desirable, especially when dealing with characters, as sign extension can lead to unexpected behavior.

A MIPS instruction that does not sign extend when loading bytes is:
- `load byte unsigned: lbu`

This instruction allows for more flexibility in handling data, particularly when it comes to loading and storing bytes, ensuring that the data is handled as unsigned when necessary.
Overflow in Arithmetic (1/2)

• Reminder: Overflow occurs when there is a “mistake” in arithmetic due to the limited precision in computers.

• Example (4-bit unsigned numbers):

\[
\begin{array}{c}
15 & 1111 \\
+ 3 & + 0011 \\
18 & 10010
\end{array}
\]

• But we don’t have room for 5-bit solution, so the solution would be 0010, which is +2, and “wrong”.
Overflow in Arithmetic (2/2)

• Some languages detect overflow (Ada), some don’t (most C implementations)

• MIPS solution is 2 kinds of arithmetic instructs:
  • These **cause overflow to be detected**
    - add (add)
    - add immediate (addi)
    - subtract (sub)
  • These **do not cause overflow detection**
    - add unsigned (addu)
    - add immediate unsigned (addiu)
    - subtract unsigned (subu)

• Compiler selects appropriate arithmetic
  • MIPS C compilers produce *addu, addiu, subu*
Two “Logic” Instructions

• Here are 2 more new instructions

• Shift Left: \texttt{sll \$s1,$s2,2} \#s1=s2<<2
  • Store in \$s1 the value from \$s2 shifted 2 bits to the left (they fall off end), inserting 0’s on right; \texttt{<<} in C.
  • Before: \texttt{0000 0002_{hex}}
    \texttt{0000 0000 0000 0000 0000 0000 0000 0010_{two}}
  • After: \texttt{0000 0008_{hex}}
    \texttt{0000 0000 0000 0000 0000 0000 0000 1000_{two}}
  • What arithmetic effect does shift left have?

• Shift Right: \texttt{srl} is opposite shift; \texttt{>>}
Loops in C/Assembly (1/3)

• Simple loop in C; \( A[\] \) is an array of ints
  
  ```
  do { 
      g = g + A[i];
      i = i + j;
  } while (i != h);
  ```

• Rewrite this as:
  
  ```
  Loop: 
      g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
  ```

• Use this mapping:
  
  ```
  g, h, i, j, &A[0]
  $s1, $s2, $s3, $s4, $s5
  ```
Loops in C/Assembly (2/3)

• Final compiled MIPS code:

```
Loop:  sll  $t1,$s3,2  # $t1= 4*i
       addu $t1,$t1,$s5  # $t1=addr A+4i
       lw   $t1,0($t1)  # $t1=A[i]
       addu $s1,$s1,$t1  # g=g+A[i]
       addu $s3,$s3,$s4  # i=i+j
       bne  $s3,$s2,Loop  # goto Loop
           # if i!=h
```

• Original code:

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```
Loops in C/Assembly (3/3)

• There are three types of loops in C:
  • while
  • do ... while
  • for

• Each can be rewritten as either of the other two, so the method used in the previous example can be applied to these loops as well.

• Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch.
Inequalities in MIPS (1/4)

• Until now, we’ve only tested equalities (== and != in C). General programs need to test < and > as well.

• Introduce MIPS Inequality Instruction:
  • “Set on Less Than”
  • Syntax:     slt reg1,reg2,reg3
  • Meaning:    reg1 = (reg2 < reg3);

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

“set” means “change to 1”,
“reset” means “change to 0”. 

Same thing...
Inequalities in MIPS (2/4)

• How do we use this? Compile by hand:
  
  \[\text{if } (g < h) \text{ goto } \text{Less}; \ #g:\$s0, \ h:\$s1 \]

• Answer: compiled MIPS code…

  \[
  \text{slt } $t0,$s0,$s1 \ # \ $t0 = 1 \text{ if } g<h \\
  \text{bne } $t0,$0,Less \ # \ \text{goto } \text{Less} \\
  \text{          } \ # \ \text{if } $t0!=0 \\
  \text{          } \ # \ (if \ (g<h)) \ \text{Less:}
  \]

• Register $0$ always contains the value 0, so \text{bne} and \text{beq} often use it for comparison after an \text{slt} instruction.

• A \text{slt} \rightarrow \text{bne} pair means \text{if}(\ldots < \ldots)\text{goto}…
Inequalities in MIPS (3/4)

- Now we can implement $<$, but how do we implement $>$, $\leq$ and $\geq$?

- We could add 3 more instructions, but:
  - **MIPS goal: Simpler is Better**

- Can we implement $\leq$ in one or more instructions using just `slt` and branches?
  - What about $>$?
  - What about $\geq$?
# Inequalities in MIPS (4/4)

```
# a:$s0, b:$s1
slt $t0,$s0,$s1  # $t0 = 1 if a < b
beq $t0,$0,skip  # skip if a >= b
    <stuff>  # do if a < b
skip:
```

**Two independent variations possible:**

*Use* `slt $t0,$s1,$s0` *instead of*

`slt $t0,$s0,$s1`

*Use* `bne` *instead of* `beq`
Immediates in Inequalities

• There is also an immediate version of `slt` to test against constants: `slti`
  • Helpful in for loops

\[
\text{C} \\
\text{if (g } \geq \text{ 1) goto Loop}
\]

\[
\text{MIPS} \\
\text{Loop: . . .} \\
\text{slti } t0, s0, 1 \quad \# \ t0 = 1 \text{ if } s0<1 \ (g<1) \\
\text{beq } t0, 0, \text{Loop} \quad \# \ \text{goto Loop} \\
\quad \# \ \text{if } t0==0 \\
\quad \# \ (if \ (g}\geq\text{1))
\]

An `slt` ↦ `beq` pair means if(... ≥ ...)goto...
What about unsigned numbers?

• Also **unsigned** inequality instructions:

  ```
  sltu, sltiu
  ```

  ...which sets result to 1 or 0 depending on unsigned comparisons

• What is value of $t0$, $t1$?

  \[
  (s0 = \text{FFFF } \text{FFFF}_{\text{hex}}, s1 = \text{0000 } \text{FFFF}_{\text{hex}}) \\
  \text{slt } t0, s0, s1 \\
  \text{sltu } t1, s0, s1
  \]
MIPS Signed vs. Unsigned – diff meanings!

• MIPS terms Signed/Unsigned “overloaded”:
  • Do/Don't sign extend
    ▷ (lb, lbu)
  • Do/Don't overflow
    ▷ (add, addi, sub, mult, div)
    ▷ (addu, addiu, subu, multu, divu)
  • Do signed/unsigned compare
    ▷ (slt, slti/sltd, sltiu)
Peer Instruction

Loop:  

\texttt{addi \$s0 \$s0}  \quad \# \ i = i - 1  \\
\texttt{slti \$t0 \$s1}  \quad \# \ \$t0 = (j < 2)  \\
\texttt{beq \$t0 \$0 Loop}  \quad \# \ \text{goto Loop if} \ \$t0 == 0  \\
\texttt{slt \$t0 \$s1 \$s0}  \quad \# \ \$t0 = (j < i)  \\
\texttt{bne \$t0 \$0 Loop}  \quad \# \ \text{goto Loop if} \ \$t0 \neq 0

(a) $j < 2 \land j < i$  
(b) $j < 2 \land j \geq i$  
(c) $j > 2 \land j < i$  
(d) $j \geq 2 \lor j < i$  
(e) $j > 2 \lor j < i$

What C code properly fills in the blank in loop below?

\texttt{do \{i--;\} while(\___);}
“And in conclusion…”

- To help the **conditional branches** make decisions concerning inequalities, we introduce: “Set on Less Than” called
  \texttt{slt, slti, sltu, sltiu}

- One can store and load (signed and unsigned) \texttt{bytes} as well as words with \texttt{lb, lbu}

- Unsigned add/sub \texttt{don’t cause overflow}

- New MIPS Instructions:
  \texttt{sll, srl, lb, lbu}
  \texttt{slt, slti, sltu, sltiu}
  \texttt{addu, addiu, subu}