Computer Systems Architecture

http://cs.nott.ac.uk/∼txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 07: Signedness, Overflow,
Multiplication and Division

Signed and Unsigned Instructions

- MIPS can interpret words as signed or unsigned
- Many instructions have signed and unsigned variants

`slt` **vs** `sltu` *dst*, src_0, src_1 – Set on Less-Than

- `slt` interprets src_0 and src_1 as *signed* integers
- While `sltu` interprets src_0 and src_1 as *unsigned*
- What is the result from each of the following instructions?

\$s1	$\mathrm{FFFFFFFF}_{16}$
\$s2	00000001_{16}

```
slt  $s0, $s1, $s2
sltu $s0, $s1, $s2
```

`slt` $\boxed{\$s0 = 1}$ because $-1 < 1$

`sltu` $\boxed{\$s0 = 0}$ because $2^{32} - 1 \not< 1$

Sign Extension

- What about signed bytes and half-words (in memory)?
- Run: `lbu $s0, ($a0)` with $\boxed{\texttt{M[\$a0]} = \texttt{123456FF}_{16}}$
  - Loads the byte $\texttt{FF}_{16}$ $(= -1_{10})$ into `$s0`
  - But now `$s0` contains $\texttt{000000FF}_{16} = 255_{10}$!
  - How do we preserve the intended value?

Sign Extension

- For a signed byte, copy the MSB (bit 7) 24 times:

$$\begin{array}{|c|c|}\hline x \quad \cdots \quad \leftarrow \quad \cdots \quad x & xyyy\ yyyy \\ \hline \multicolumn{1}{c}{\leftarrow \quad 24 \text{ bits} \quad \rightarrow} & \multicolumn{1}{c}{\leftarrow 8 \text{ bits} \rightarrow}\end{array}$$

- e.g. $\texttt{D6}_{16}$ $(= -42_{10})$ is sign-extended to $\texttt{FFFFFFD6}_{16}$
- `lb` and `lh` performs *sign extension*; `lbu` and `lhu` does not
  - Usually use bytes for characters, so `lbu` used more often

Signed Overflow

- `add`/`addi`/`addu`/`addiu` use the same addition circuits
  - But constants in `addi`/`addiu` are always sign-extended
- However, `add` and `addi` also check for overflow
  - Overflow when result exceeds $-2^{31} \le x < 2^{31}$
  - On overflow, trigger an error exception
  - Try `li $t0, 0x7fffffff` in SPIM!

    ```
    li $t0, 0x7fffffff
    addi $t0, $t0, 1
    ```
- `addu`/`addiu` doesn't check for overflow
  - We can check for ourselves: avoid triggering exception
- There is also `subu`, subtraction without overflow checking

Checking for Overflow

```
$dst$ = $src_0$ + $src_1$
if(sign($src_0$) != sign($src_1$))
    goto no_overflow;
if(sign($dst$) == sign($src_0$))
    goto no_overflow;
    # we have overflow!
no_overflow:
# rest of program
```

```
addu $s0, $s1, $s2
xor $t0, $s1, $s2
blt $t0, $zero, no_overflow
xor $t0, $s0, $s1
bge $t0, $zero, no_overflow
    # we have overflow!
no_overflow:
# rest of program
```

- `xor` dst, src_0, src_1 returns
  - a positive dst when the sign bit of src_0 and src_1 match
  - a negative dst when the sign bit of src_0 and src_1 differ
- `sign`(x)=x&0x8000 for 32 bit numbers.

Multiplication

- Product of *m*- and *n*-digit numbers requires $m + n$ digits
    - Multiplying 4-digit numbers needs 8 digits
- Binary case needs only multiply by 0 or 1, and addition

Long Multiplication in Decimal

$$
\begin{array}{cccccccccl}
 & & & & & 6 & 2 & 9 & 5 & \\
\times & & & & & 2 & 8 & 1 & 7 & \\
\hline
 & & & & 4 & 4 & 0 & 6 & 5 & \\
 & & & & 6 & 2 & 9 & 5 & & \\
 & & 5 & 0 & 3 & 6 & 0 & & & \\
+ & 1 & 2 & 5 & 9 & 0 & & & & \text{Partial Sums} \\
\hline
= & 1 & 7 & 7 & 3 & 3 & 0 & 1 & 5 &
\end{array}
$$

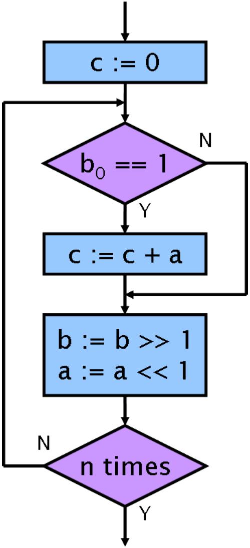
Multiplication

- Product of m- and n-digit numbers requires $m + n$ digits
  - Multiplying 4-digit numbers needs 8 digits
- Binary case needs only multiply by 0 or 1, and addition

Long Multiplication in Binary

$$
\begin{array}{ccccccccc l}
 & & & & & 1 & 1 & 0 & 1 & \mathrm{a} = 13_{10} \\
\times & & & & & 1 & 0 & 1 & 1 & \mathrm{b} = 11_{10} \\
\hline
 & & & & & 1 & 1 & 0 & 1 & \\
 & & & & 1 & 1 & 0 & 1 & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
+ & & 1 & 1 & 0 & 1 & & & & \text{Partial Sums} \\
\hline
= & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \mathrm{c} = 143_{10}
\end{array}
$$

n-Bit Binary Multiplication



- Given a and b, calculates c = a * b
- Optimisation: exit as soon as b == 0
- But doesn't work for signed numbers!
    - Take magnitude, multiply, then fix sign?
- Equivalent C code:

```
c = 0;
for(i = 1; i < n; i = i + 1) }
    if(b & 0x01 != 0)
        c = c + a;
    b = b >> 1;
    a = a << 1;
    }
```

Signed Binary Multiplication

- Signed numbers can be infinitely sign-extended
  - Positive numbers prefixed by an ‘infinite’ number of 0s
  - Negative numbers prefixed by an ‘infinite’ number of 1s
- Sign-extend *a* and *b* to 2*n* digits for multiplication
  - Must loop 2*n* times as a result
  - But can still exit early when `b == 0`

Example: -3×5

$$
\begin{array}{cccccccccccl}
 & \ldots & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & & \mathrm{a} = -3_{10} \\
\times & \ldots & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & & \mathrm{b} = 5_{10} \\
\hline
 & \ldots & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & & \\
 & \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
+ & \ldots & 1 & 1 & 1 & 1 & 0 & 1 & & & & \text{Partial Sums} \\
\hline
= & \ldots & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & & \mathrm{c} = 15_{10}
\end{array}
$$

Division

Long Division in Decimal

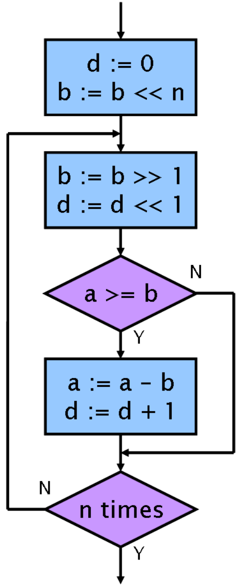
	Divisor		0	0	2	1	3	Quotient
÷	8	1	1	7	3	3	0	Dividend
		8	1	0	0	0	0	× 0
			8	1	0	0	0	× 0
				8	1	0	0	× 2
		−	1	6	2	0	0	
				1	1	3	0	
					8	1	0	× 1
				−	8	1	0	
					3	2	0	
						8	1	× 3
				−	2	4	3	
						7	7	Remainder

Division

Long Division in Binary

		Divisor		0	0	1	0	1	1	1	Quotient
÷	1	0	1	1	1	1	0	1	0	1	Dividend
		1	0	1	0	0	0	0	0	0	× 0
			1	0	1	0	0	0	0	0	× 0
			−	1	0	1	0	0	0	0	× 1
					1	0	0	1	0	1	
					1	0	1	0	0	0	× 0
					−	1	0	1	0	0	× 1
						1	0	0	0	1	
						−	1	0	1	0	× 1
								1	1	1	
							−	1	0	1	× 1
									1	0	Remainder

n-Bit Binary Division



- Given dividend a and divisor b
  - Calculates their quotient d = a / b
  - Leaves the remainder in a = a % b
- Equivalent C code:

```
d = 0;
b = b << n;
for(i = 0; i < n; i++) {
    b = b >> 1;
    d = d << 1;
    if(a >= b) {
        a = a - b;
        d = d + 1;
        }
    }
```

Multiplication and Division on the MIPS

- Mul$^{\underline{n}}$ produces a 64-bit word; div$^{\underline{n}}$ two 32-bit results
  - No way to encode two destination registers. . .
- Slow compared to e.g. addition; takes many cycles
  - Would stall the next instructions in the *pipeline*
- Independent unit (from main ALU) for mul$^{\underline{n}}$ and div$^{\underline{n}}$
  - Source operands come from the usual register file
  - Results written to two special registers `HI` and `LO`

`mfhi` *dst* / `mflo` *dst* — move from `HI`/`LO`

- `mfhi` *dst* — *dst* := `HI`
- `mflo` *dst* — *dst* := `LO`

MIPS Multiplication

mult src_0, src_1 / multu src_0, src_1 — multiplication

- `HI` := upper 32 bits of $src_0 * src_1$
  `LO` := lower 32 bits of $src_0 * src_1$
- `mult` treats src_0/src_1 as signed; `multu` as unsigned

mul dst, src_0, src_1 — multiplication (no overflow check)

- dst := `LO` := lower 32 bits of $src_0 * src_1$
- Single instruction equivalent of
  ```
  mult src0, src1
  mflo dst
  ```
- No `mulu` – same result signed or unsigned

Multiplication Overflow

- Pseudoinstructions `mulo` and `mulou` check for overflow
  - Result too large for a 32-bit signed/unsigned word
  - How do these pseudoinstructions work?
- Replace `break $0` with your own error-handing code

mulou *dst*, src_0, src_1

```
mult src0, src1
mfhi $at


beq $at, $0, no_overflow
    break $0
no_overflow:
mflo dst
```

mulo *dst*, src_0, src_1

```
mult src0, src1
mfhi $at
mflo dst
sra dst, dst, 31
beq $at, dst, no_overflow
    break $0
no_overflow:
mflo dst
```

MIPS Division

div src_0, src_1 / divu src_0, src_1 — division

- `HI := `src_0 % src_1
  `LO := `$src_0 \div src_1$
- `div` treats src_0/src_1 as signed; `divu` as unsigned

div dst, src_0, src_1 / divu dst, src_0, src_1 — division

- Three argument pseudoinstruction version of `div`/`divu`
- Expands to `div` src_0, src_1 or `divu` src_0, src_1
  `mflo` dst `mflo` dst