

NUMERICAL OPERATIONS

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

Overview

- This lecture
 - ▣ Overflow for addition/subtraction
 - ▣ Multiplication
 - ▣ Division

Addition/Subtraction Overflow

- **Unsigned Numbers:** overflow happens when the last carry (1) cannot be accommodated.
 - ▣ i.e., there is not enough memory bits to represent the number

Example:

$$\begin{array}{r} 7_{\text{dec}}: 0111_{\text{bin}} \\ + 10_{\text{dec}}: 1010_{\text{bin}} \\ \hline \end{array}$$

$$\begin{array}{r} 7_{\text{dec}}: 0111_{\text{bin}} \\ + 2_{\text{dec}}: 0010_{\text{bin}} \\ \hline \end{array}$$

Addition/Subtraction Overflow

- **Unsigned Numbers:** overflow happens when the last carry (1) cannot be accommodated.
 - ▣ i.e., there is not enough memory bits to represent the number

Example:

$$\begin{array}{r} 7_{\text{dec}}: 0111_{\text{bin}} \\ + 10_{\text{dec}}: 1010_{\text{bin}} \\ \hline 1_{\text{dec}}: 0001_{\text{bin}} \end{array}$$

$$\begin{array}{r} 7_{\text{dec}}: 0111_{\text{bin}} \\ + 2_{\text{dec}}: 0010_{\text{bin}} \\ \hline 9_{\text{dec}}: 1001_{\text{bin}} \end{array}$$

Addition/Subtraction Overflow

- **Signed Numbers:** overflow happens if the most significant bit is not the same as every bit to its left
 - e.g., the sum of two positive numbers is a negative result
 - e.g., the sum of two negative numbers is a positive result
- ▣ Adding positive and negative numbers will not overflow

Example:

$$\begin{array}{r} +7_{\text{dec}}: 0111_{\text{bin}} \\ + -6_{\text{dec}}: 1010_{\text{bin}} \\ \hline \end{array}$$

$$\begin{array}{r} +7_{\text{dec}}: 0111_{\text{bin}} \\ + +2_{\text{dec}}: 0010_{\text{bin}} \\ \hline \end{array}$$

Addition/Subtraction Overflow

- **Signed Numbers:** overflow happens if the most significant bit is not the same as every bit to its left
 - e.g., the sum of two positive numbers is a negative result
 - e.g., the sum of two negative numbers is a positive result
- ▣ Adding positive and negative numbers will not overflow

Example:

$$\begin{array}{r} +7_{\text{dec}}: 0111_{\text{bin}} \\ + \\ -6_{\text{dec}}: 1010_{\text{bin}} \\ \hline +1_{\text{dec}}: 0001_{\text{bin}} \end{array}$$

$$\begin{array}{r} +7_{\text{dec}}: 0111_{\text{bin}} \\ + \\ +2_{\text{dec}}: 0010_{\text{bin}} \\ \hline -7_{\text{dec}}: 1001_{\text{bin}} \end{array}$$

MIPS Instructions

- Instructions **add**, **addi**, and **sub** may cause **exceptions** on overflow
 - ▣ Software needs to handle exceptions
 - **More on this later**
- MIPS provides the **addu**, **addiu**, and **subu** instructions that work with unsigned integers and never flag an overflow
 - ▣ Other instructions may be executed to detect the overflow

Dealing with >32 -bit

- Example 1: Store the unsigned value 6538305685_{dec} in the register file.

1 10000101 10110110 10100000 10010101_{bin}

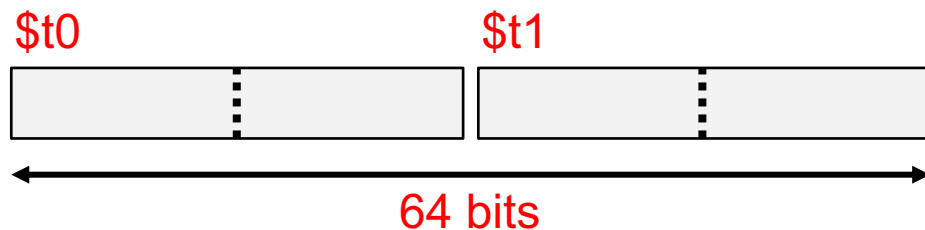
1 85B6 A095_{hex}

Dealing with >32 -bit

- Example 1: Store the unsigned value 6538305685_{dec} in the register file.

1 10000101 10110110 10100000 10010101_{bin}

1 85B6 A095_{hex}

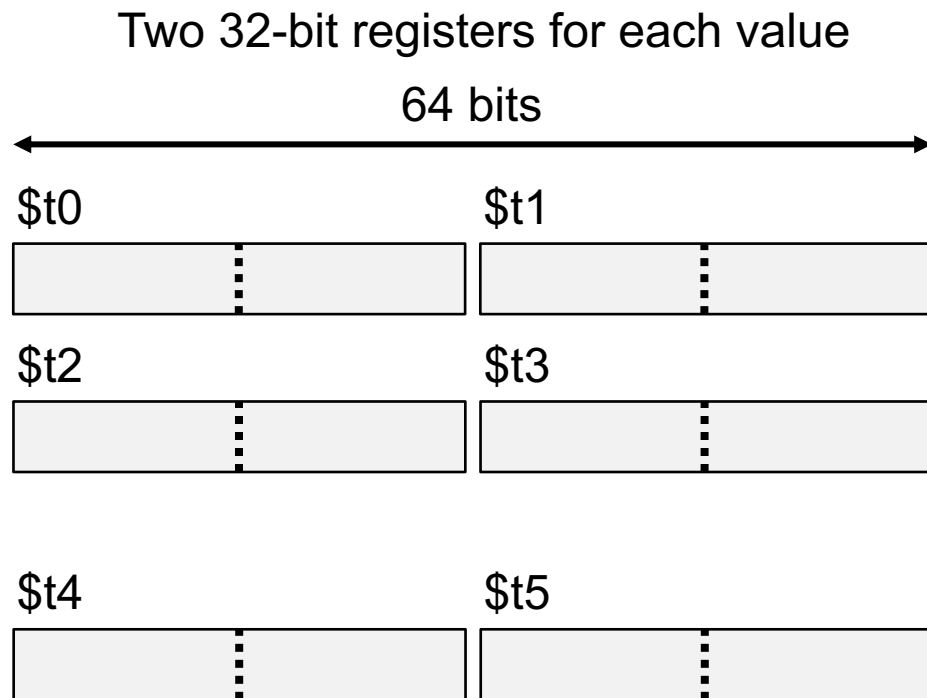


Two 32-bit registers (\$t0 and \$t1)

```
lui $t1, 0x85B6
ori $t1, $t1, 0xA095
ori $t0, $zero, 1
```

Dealing with >32 -bit

- Example 2: add the unsigned 64-bit values in ($\$t0, \$t1$) and ($\$t2, \$t3$). Store the result in ($\$t4, \$t5$).



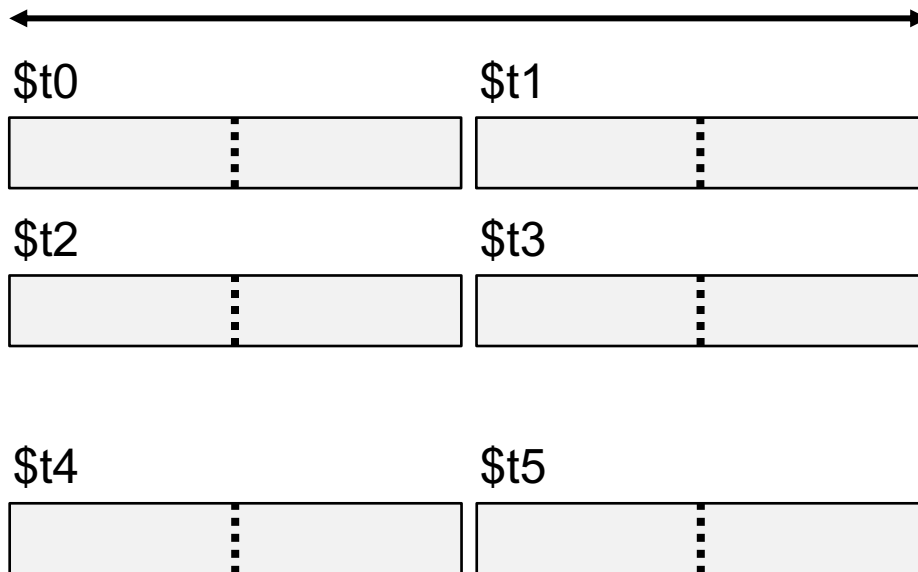
```
addu $t5, $t1, $t3  
addu $t4, $t0, $t2
```

Dealing with >32 -bit

- Example 2: add the unsigned 64-bit values in ($\$t0, \$t1$) and ($\$t2, \$t3$). Store the result in ($\$t4, \$t5$).

Two 32-bit registers for each value

64 bits



```
addu $t5, $t1, $t3
addu $t4, $t0, $t2
sltu $t6, $t5, $t1
sltu $t7, $t5, $t3
beq $t6, $zero, OK
beq $t7, $zero, OK
addiu $t4, $t4, 1
```

OK: ...

Multiplication Example

- Multi-step process
- Every step
 - ▣ multiplicand is shifted
 - ▣ next bit of multiplier is examined (also a shifting step)
 - ▣ if this bit is 1, shifted multiplicand is added to the product

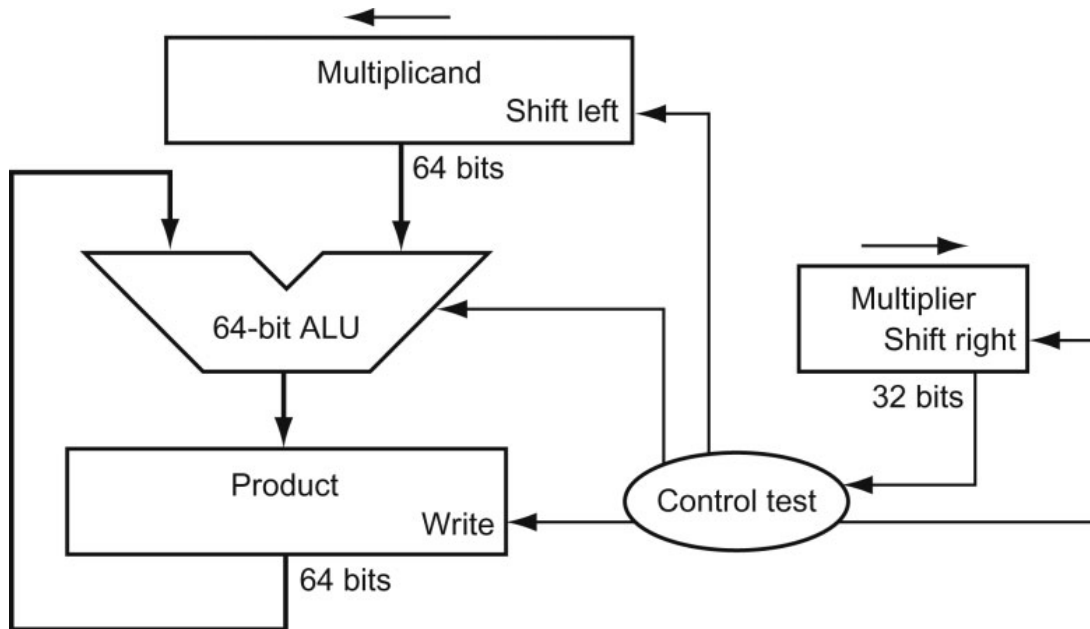
Multiplicand		1000 _{ten}
Multiplier	x	1001 _{ten}

		1000
		0000
		0000
		1000

Product		1001000 _{ten}

Multiplication Example

- Multi-step process



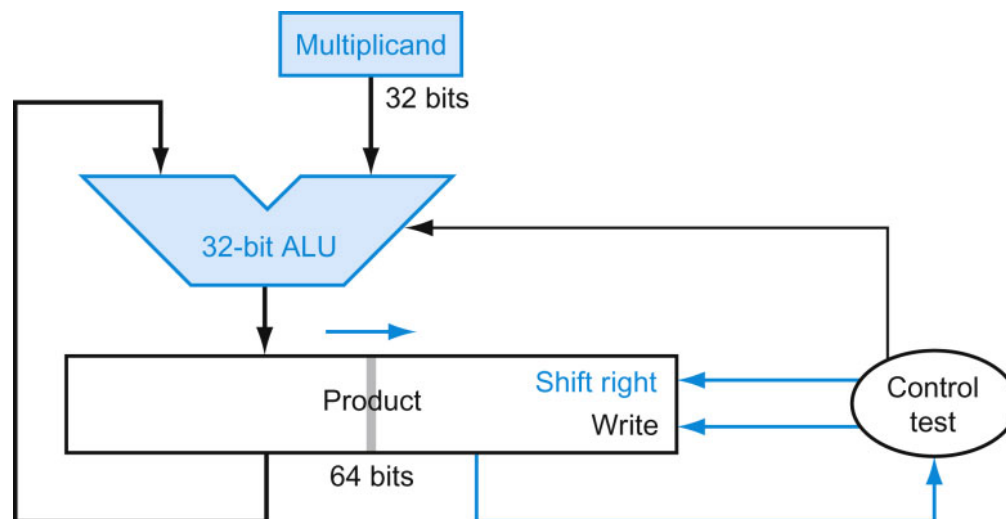
Multiplicand 1000_{ten}
Multiplier $\times 1001_{\text{ten}}$

 1000
 0000
 0000
 1000

Product 1001000_{ten}

Multiplication Algorithm 2

- A more efficient algorithm
 - ▣ 32-bit ALU and multiplicand is untouched
 - ▣ sum keeps shifting right
 - number of bits in product + multiplier = 64,
 - hence, they share a single 64-bit register



Multiplication Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number
 - ▣ In MIPS, the product is saved in two 32-bit registers

MIPS Instructions

□ Signed multiplication (mult)

`mult` `$s2, $s3` computes the product and stores it in two “internal” registers that can be referred to as `hi` and `lo`

`mfhi` `$s0` moves the value in `hi` into `$s0`

`mflo` `$s1` moves the value in `lo` into `$s1`

□ Similarly for unsigned multiplication (multu)

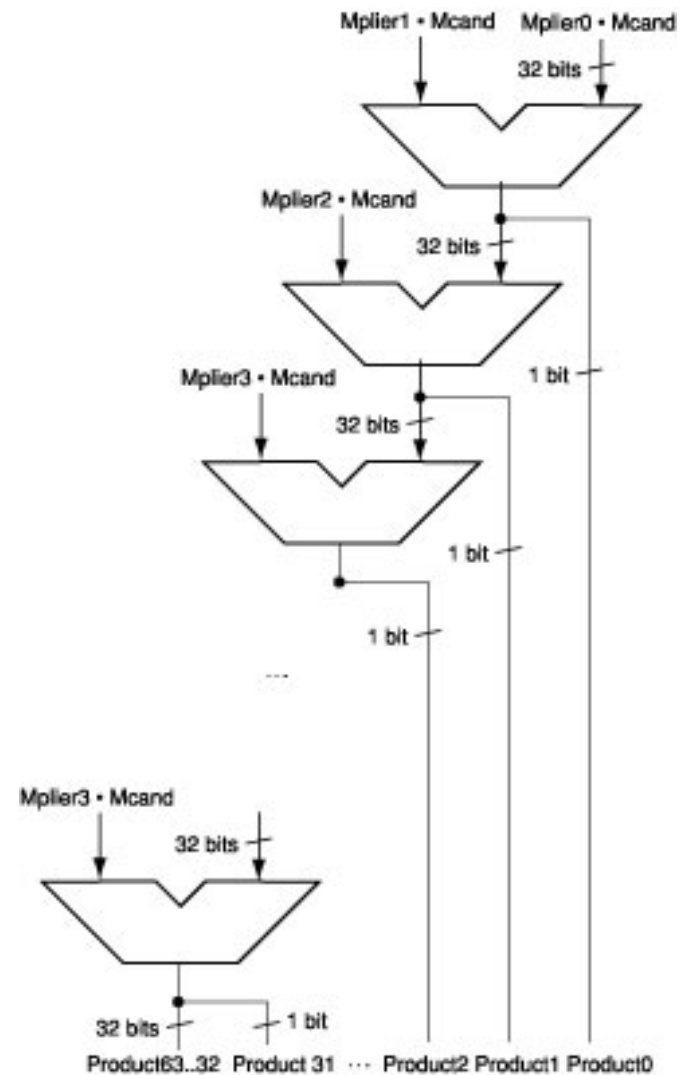
`multu` `$s2, $s3`

`mfhi` `$s0`

`mflo` `$s1`

Multiplication: Fast Algorithm

- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting
- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved
- **Note:** high transistor cost



Division Example

- Multi-step process
 - ▣ shift divisor right and compare it with current dividend
 - if divisor is larger, shift 0 as the next bit of the quotient
 - if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

		$\begin{array}{r} 1001_{\text{ten}} \\ \hline 1001010_{\text{ten}} \\ -1000 \\ \hline 10 \\ 101 \\ 1010 \\ -1000 \\ \hline 10_{\text{ten}} \end{array}$	Quotient
Divisor	1000_{ten}		Dividend
			Remainder

Division Example

- Divide 7_{ten} (0000 0111_{two}) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

Division Example

- Divide 7_{ten} ($0000\ 0111_{\text{two}}$) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem ≥ 0 → shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001