EE382A – Spring 2025

Chapter 7:
Kestrel’s Instruction Set Architecture (4 of 5)

Kestrel's ISA, fourth part

- Single-precision multiplication
- Multi-precision multiplication
- Optional programming assignment: fixed-point multiplication
- Bit-shifting
- Deeply nested conditionals

How does multiplication work?

Single-precision multiplication

MULT Returns low byte of product of two unsigned bytes

```
mult     Rdest, OpA, OpB
```

MULTSA Returns low byte of product of a signed byte (first operand, OpA) and an unsigned byte (second operand, OpB)

```
multsa   Rdest, OpA, OpB
```

MULTSB Returns low byte of product of an unsigned byte (first operand, OpA) and a signed byte (second operand, OpB)

```
multsb   Rdest, OpA, OpB
```

MULTSAB Returns low byte of product of a signed byte (first operand, OpA) and a signed byte (second operand, OpB)

```
multsab  Rdest, OpA, OpB
```

The **`MHI`** register contains the product's high byte.

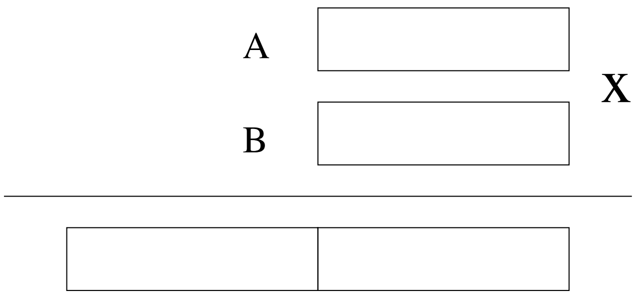
Single-precision multiplication (cont.)

NOTE: the signed operations are only performed when they involve the most-significant byte (MSB) of a signed operand, NOT on all the bytes of a signed operand!

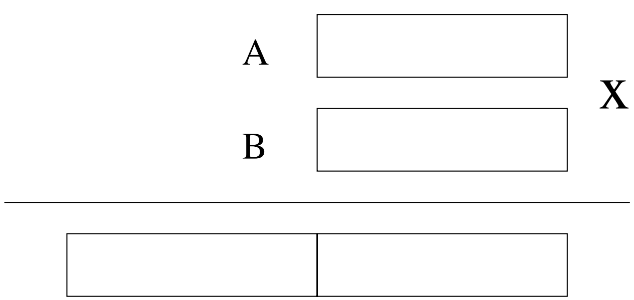
SMHI is the sign extension of the multiplier high byte, **`mhi`**

Single-precision multiplication: examples

Example 1: both operands unsigned

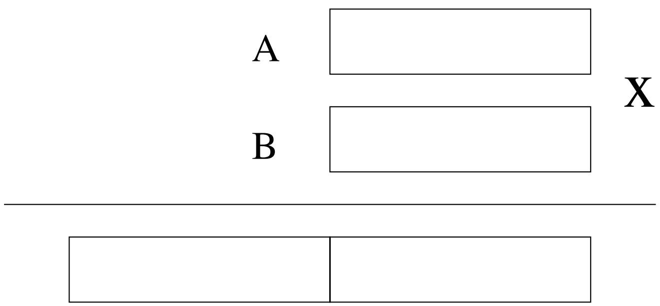


Example 2: A signed, B unsigned

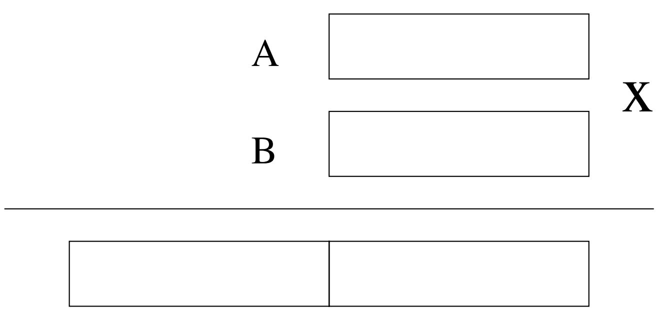


Single-precision multiplication: more examples

Example 3: A unsigned, B signed



Example 4: both A and B signed



Multi-precision multiplication

- How does it work?

Multi-precision multiplication

With MULT-type instructions, use the modifiers

ADDMC to add the unsigned OpC to the product

```
addmc   OpC
```

ADDMHI to add unsigned register **`mhi`** to the product

```
addmhi
```

Note that they can both be used in the same instruction.

Multiply-accumulate: operands

Kestrel can perform a fused multiply-add in one clock cycle, using the instruction:

```
addmc  OpC
```

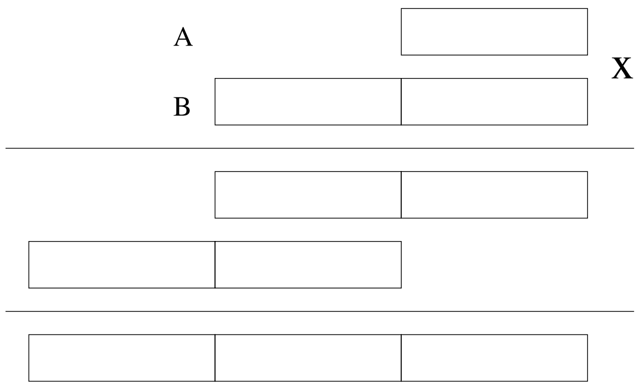
The accumulator for the addition must be a register, so the second source operand (OpB) cannot come from the SSR. The strategy is to read the multiplier from memory. Example:

```
mult  L10, L0, mdr, addmc L20
```

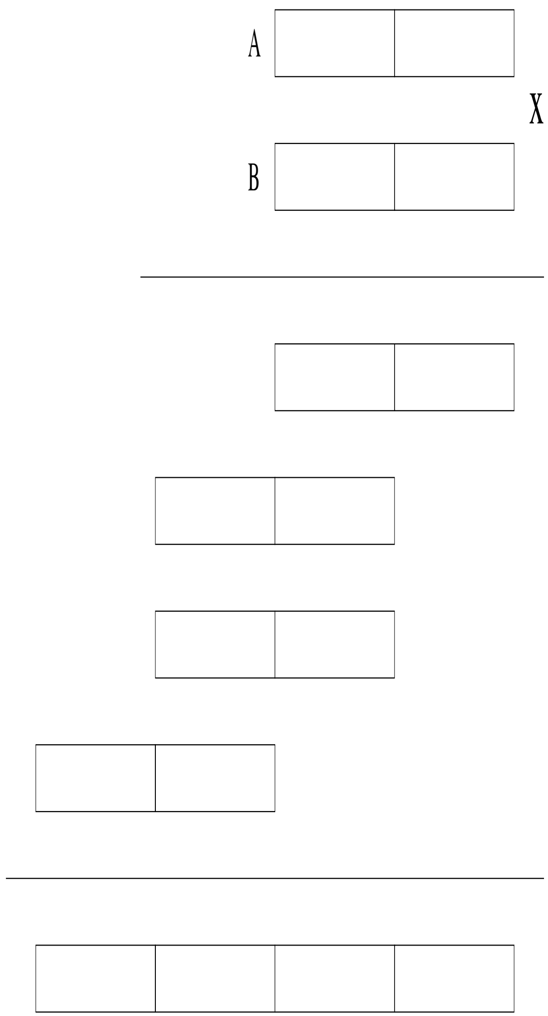
multiplies register **`L0`** by the **`mdr`**, adds register **`L20`** to the product, and stores the sum into register **`L10`**.

Multi-precision multiplication examples

Both operands unsigned



Multi-precision multiplication: unsigned



Multiply-accumulate

Note that neither **`addmc`** nor **`addmhi`** set or use the **`mp`** flag:

- the **`mp`** flag is local to the ALU;
- such a flag is not needed in the multiplier. Why?

Multiplication's actual operands

When we write, for example:

```
mult  L10, L0, mdr, addmc L20
```

register **L0** is OpA and the **mdr** is OpB.

However, it would be the same even if we swapped them, as in:

```
mult  L10, mdr, L0, addmc L20
```

since the **mdr** can only be OpB the assembler swaps them automatically. In this case it would not make any difference, but it would with signed multiplication. In this instruction:

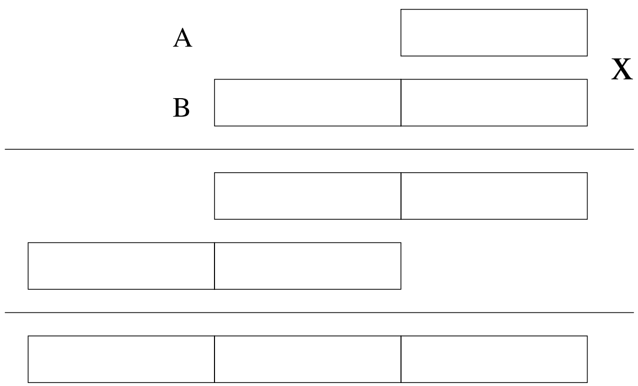
```
multsb  L4, mdr, L10, addmc L20
```

it is **L10** that is considered signed, not the **mdr.**

Always write the operands as OpA, OpB to avoid confusion!

Signed Multi-precision multiplication

A signed, B unsigned



Use the signed operations only on the MSB of each signed operand.

When using signed operations, sign-extend the partial product.

Signed multiplication: sign extensions

In most cases (excluding some operand conflicts) the sign extensions of all registers is available by just **pre-pending the letter `S` to the register name.**

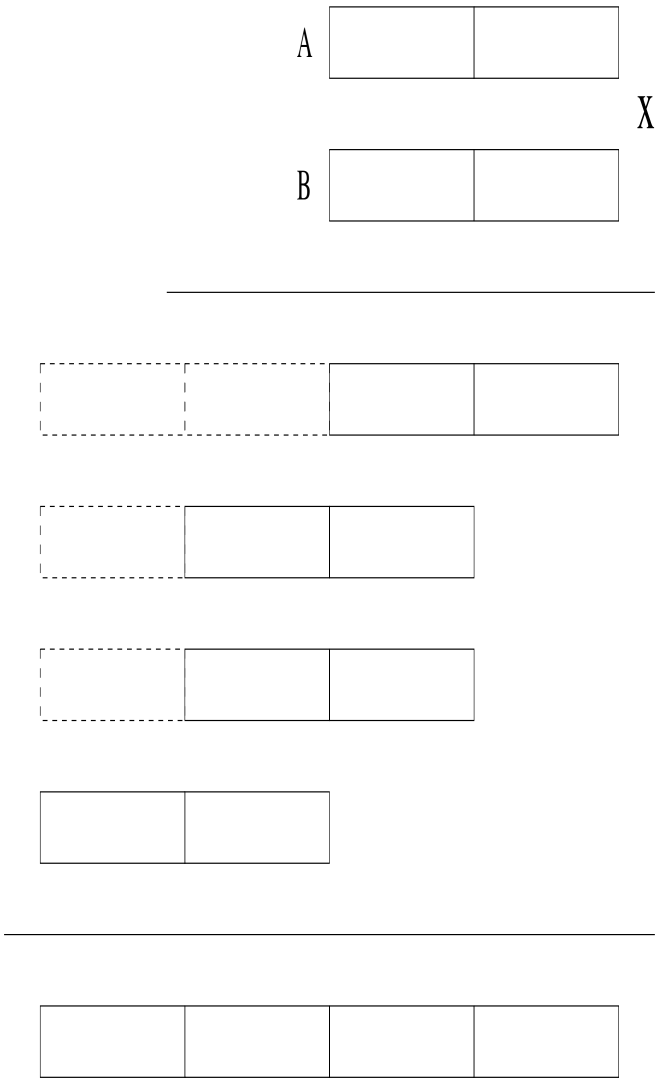
SSR registers: **`SL0`** to **`SL31`**, and **`SR0`** to **`SR31`**

MHI register: **`smhi`**

MDR register: **`smdr`**

The sign extensions are *rvalues* only.

Multi-precision multiplication: signed



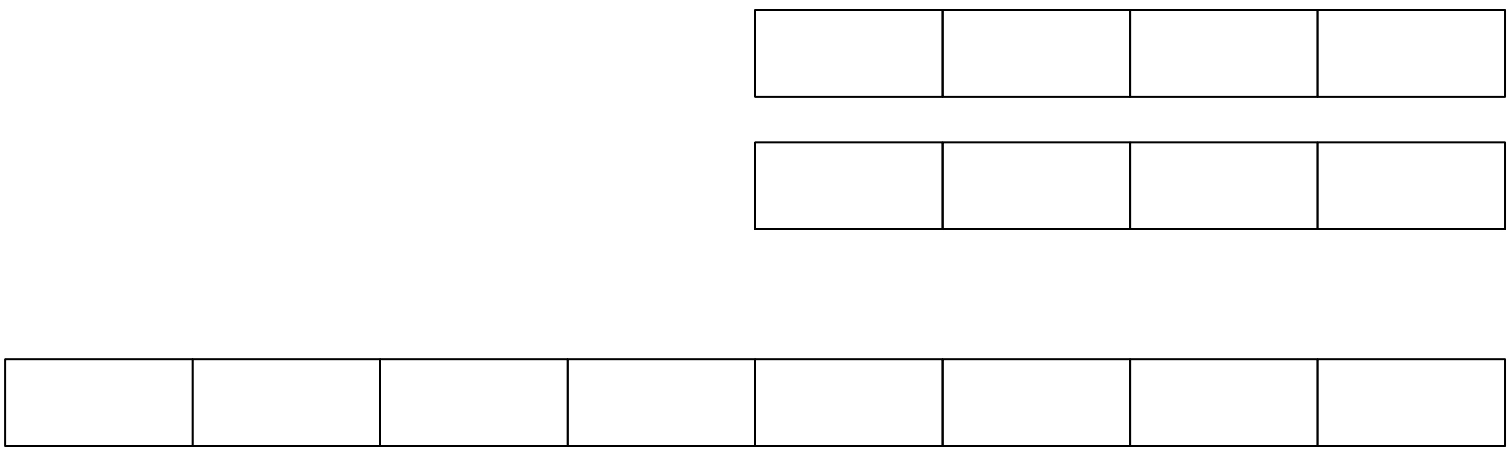
Signed multi-precision multiplication

Why use the signed operations only on the MSB of each operand?

Fixed-point representation

Used extensively in image processing and in machine learning.

Also used in our last SIMD assignment, Mandelbrot, like this:



Optional programming assignment: fixed-point multiplication

Multiply two 2-byte numbers that represent fixed-point values in the format 4.12 (4-bit integer part and 12-bit fractional part), and produce a product *in the same format*, on two bytes.

Actual shift operations in Kestrel (1/2)

In Kestrel, shifting is done with the multiplier, NOT with the bit-shifter. Example of **left shift**: [L1:L0] << 3

Actual shift operations in Kestrel (2/2)

Example of **right shift**: [L1:L0] >> 3.

How to differentiate between logic and arithmetic right shift?

Deeply nested conditionals

The bit shifter can't control nested conditionals deeper than 8 levels, so:

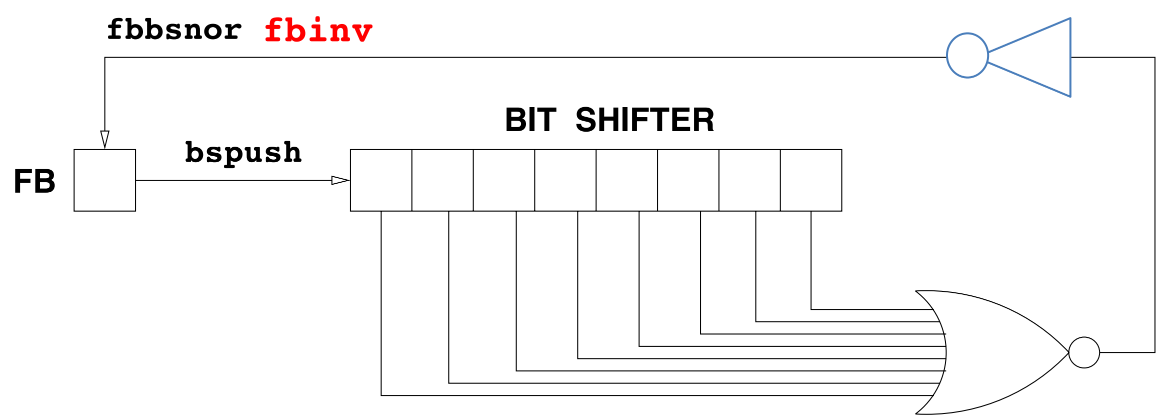
1) Save the bit shifter in a register or in memory

```
move  L29, bs
```

2) Push onto the bit shifter a bit to summarize the on/off state of the PE with **`fbbsnor fbinv`**, which writes the flag bus with the OR of all bits in the bit shifter.

```
fbbsnor fbinv bspush
```

Note that there is no need to clear the bit shifter before pushing it.



Deeply nested conditionals (cont.)

3) Restore the bit shifter when returning:

```
move  L29, L29 bslatch
```

that executes in all PEs regardless of the mask, and sets the mask.

NOTE that:

```
move  L29, L29, bscondlatch
```

only executes in active PEs and does *not* set the mask, therefore do not use it for conditional stack execution. We'll see its use later.