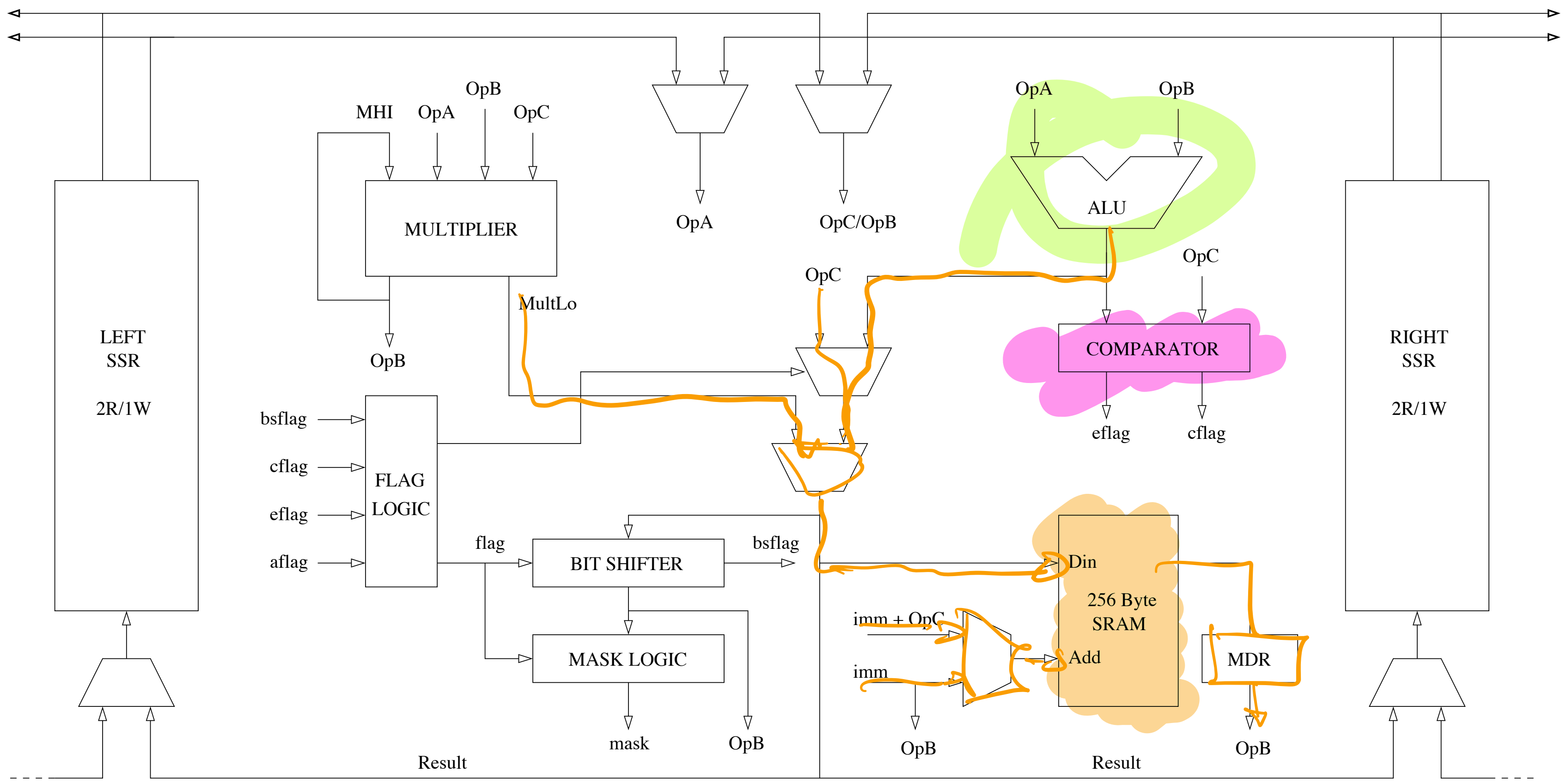
EE382A – Spring 2025

Chapter 5: Kestrel's Instruction Set Architecture (3 of 5)

Kestrel’s ISA, third part

- Bitwise logical operations
- Multi-precision addition, subtraction, selection
- Multi-precision comparison
- Controller scratch register and run-time loop counter
- Local memory

Kestrel PE diagram



Bitwise logical operations

NOP No ALU operation

```
nop
```

INVERT Bitwise inversion (1's complement)

```
invert Rdst, {OpA | OpB}
```

AND, NAND, NOR, OR, XNOR, XOR Bitwise logical operations

```
{and|nand|nor|or|xnor|xor} Rdst, OpB, OpA
```

Subtraction

SUB Difference of two operands: OpB - OpA

```
sub  Rdst, OpB, OpA
```

SUBXZ Subtract zero from operand: {OpA | OpB} - 0

```
subxz  Rdst, {OpA | OpB}
```

~~**SUBZX** Subtract operand from zero: 0 - {OpA | OpB}~~

~~`subzx  Rdst, {OpA | OpB}`~~

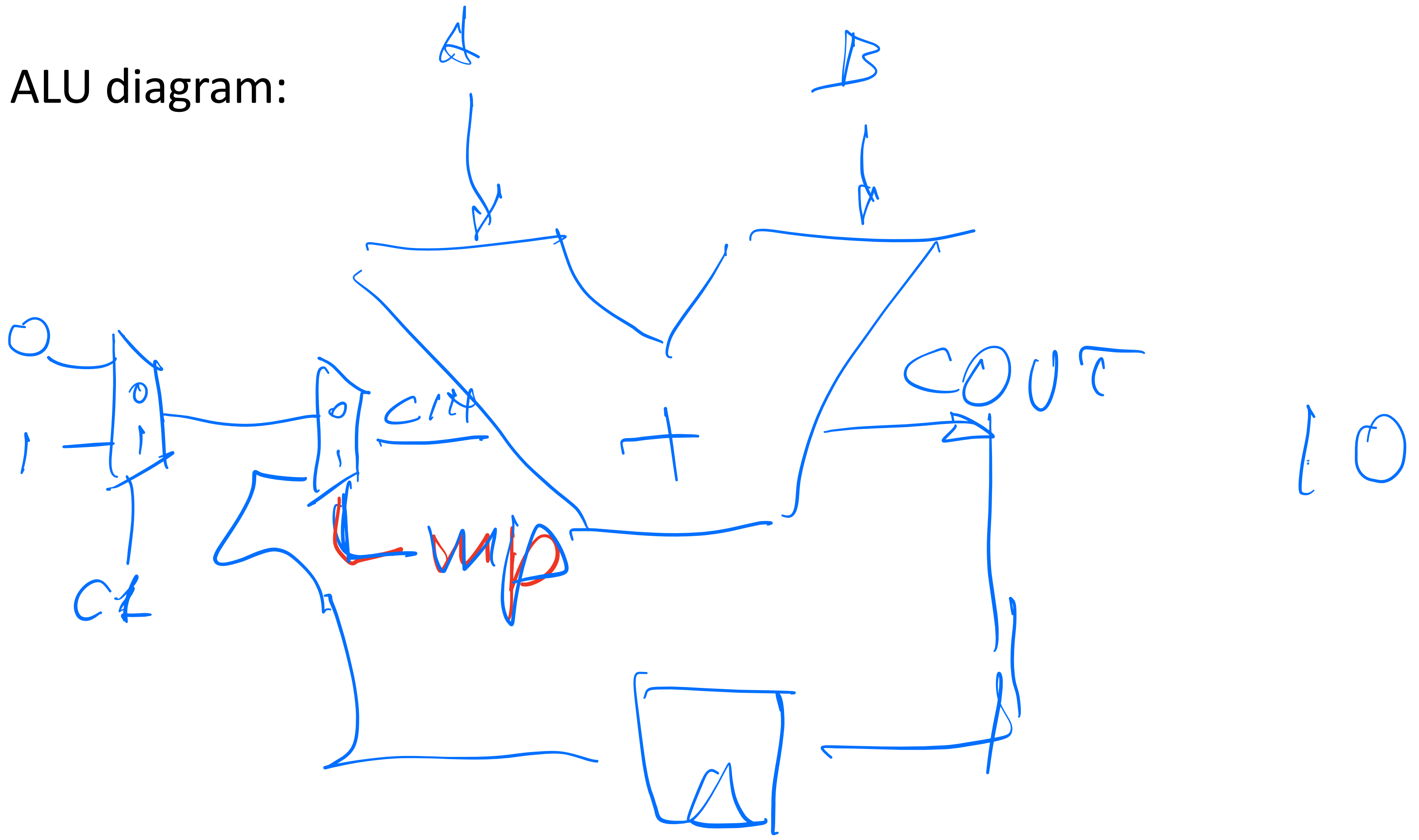
~~(needs to be fixed - it works like a **SUBZZ** now)~~

SUBZZ Subtract zero from zero: 0 - 0

```
subzz  Rdst
```

Multi-precision addition

ALU diagram:



Multi-precision addition

Use modifiers:

- **c0** Set the ALU carry-in input to zero (default).
- **c1** Set the ALU carry-in input to one.
- **mp** Set the ALU carry-in input to the carry-out of the previous operation.

Examples:

```
addzz  c0  L3       ; sets L3 = 0
addzz  c1  L3       ; sets L3 = 1
addxz  c1  L7, L7   ; increments L7 by one
```

L3 = 0 + 0 + 0

L3 = 0 + 0 + 1

L7 = L7 + 1

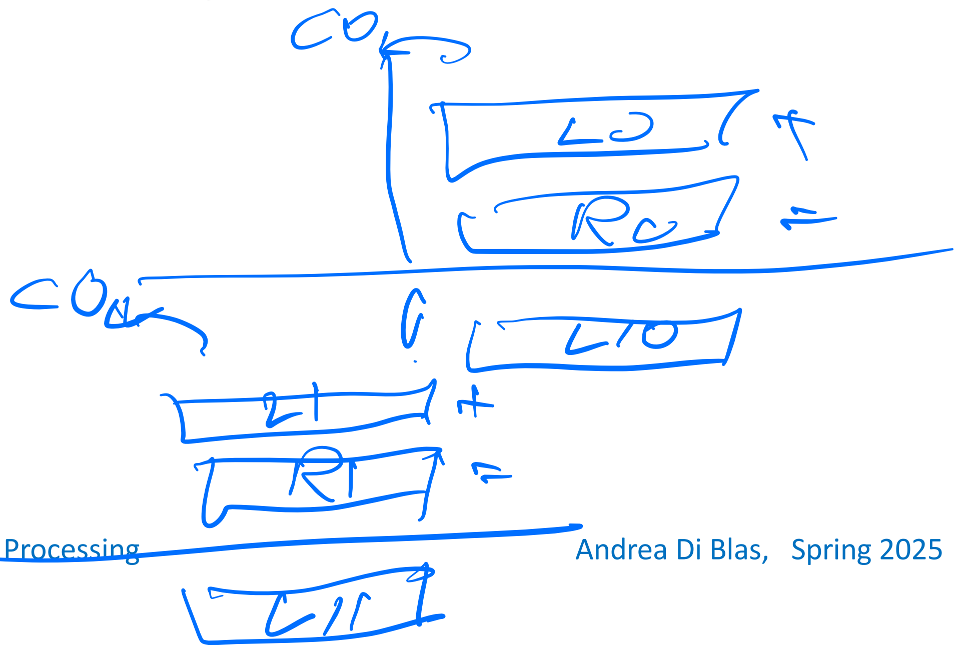
Multi-precision addition example

Assume [L1, L0] = A, [R1, R0] = B, produce [L11, L10] = A + B:

```
add          L10, L0, R0    ; Low byte
add     mp   L11, L1, R1    ; High byte
addzz   mp   L12            ; Overflow?
```

Difference between signed and unsigned: A + B where A = 20, B = -4

Overflow detection?



Addition: overflow

Unsigned: overflow = carry out (**`fbaco`**)

Signed: overflow = ALU true sign != result sign (**`fbats`**)

Example on 8 bits:

UNSIGNED

CO 1 1 1 1 1 1 1 1 0

255 ~~64~~ 1 1 1 1 1 1 1 1 +

1 ~~-48~~ 0 0 0 0 0 0 0 1 =

----

0 0 0 0 0 0 0 0

SIGNED

1 1 1 1 1 1 1 0

127 ~~-64~~ 0 0 1 1 1 1 1 1 1 +

1 ~~-96~~ 0 0 0 0 0 0 0 0 1 =

----

0 1 0 0 0 0 0 0 0 = -128

No overflow when numbers have different signs.

Addition: overflow

From RISC-V Specification, User mode:

~~We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using 18 Volume I: RISC-V Unprivileged ISA V20191213 RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: **add t0, t1, t2; bltu t0, t1, overflow**. For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: **addi t0, t1, +imm; blt t0, t1, overflow**. This covers the common case of addition with an immediate operand. For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative. **add t0, t1, t2 slti t3, t2, 0 slt t4, t0, t1 bne t3, t4, overflow**~~

Multi-precision subtraction

Example: 10 – 30 on 8 bits

b_0

```
        1  1  1  1  1
10      0  0  0  0  1  0  1  0
30      0  0  0  1  1  1  1  0
---------------------------------
-20     1  1  1  0  1  1  0  0
20      0  0  0  1  0  1  0  0
```

Multi-precision subtraction example

Assume [L1, L0] = A, [R1, R0] = B, produce [L11, L10] = A - B:

```
sub          L10, L0, R0    ; Low byte
sub     mp   L11, L1, R1    ; High byte
subzz   mp   L12            ; Overflow?
```

Difference between signed and unsigned: A - B, with A = 10, B = 12

Overflow detection?

$$A - B = A + (-B)$$

Selection: signed min and max

SMAXC Maximize with C: **`Rdst`** gets the maximum of the signed ALU result and a signed operand (**`OpC`**).

```
<ALUop> Rdst, OpA, OpB smaxc OpC or
smaxc Rdst, OpA, OpC
```

SMINC Minimize with C: **`Rdst`** gets the minimum of the signed ALU result and a signed operand (**`OpC`**).

```
<ALUop> Rdst, OpA, OpB sminc OpC or
sminc Rdst, OpA, OpC
```

Multi-precision selection

In all cases, start from the MSB, and use:

CMP with all unsigned instructions:

`maxc, minc, ~~modmaxc, modminc~~, selectc`

CMPSWAP with all signed instructions:

`smaxc, sminc`

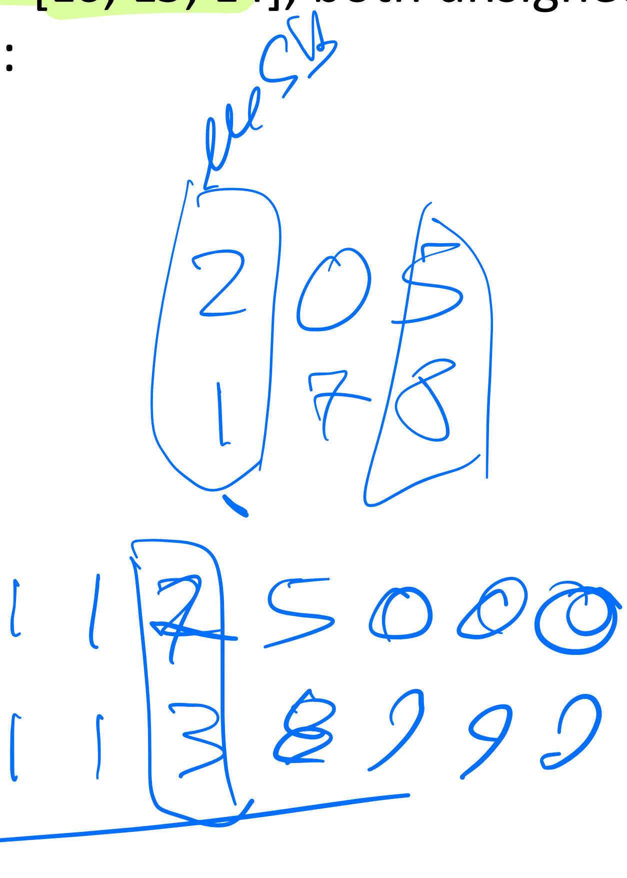
However, in the signed case the MSBs have to be swapped.

Example: unsigned multi-precision selection

To select the max of A = [L2, L1, L0] and B = [L6, L5, L4], both unsigned, storing the result in [L12, L11, L10], do:

```
maxc  L12, L2, L6
maxc  L11, L1, L5 cmp
maxc  L10, L0, L4 cmp
```

msb



Example: signed multi-precision selection

To select the max of A = [L2, L1, L0] and B = [L6, L5, L4], both signed,
storing the result in [L12, L11, L10], do:

```
move L2, #255       ; sample value: A = -23
move L1, #255
move L0, #233
move L6, #0         ; sample value: B = 200
move L5, #0
move L4, #200

smaxc   L12,  L6,  L2         ; note the swapped MSBs
smaxc   L11,  L1,  L5 cmpswap
smaxc   L10,  L0,  L4 cmpswap
```

1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 0 1 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 0 1 0 0 0

Multi-precision comparison

Compare multi-byte numbers starting from the MSB.

Use `ltc` on the MSB, and then again `ltc` with:

CMP on the other bytes, for *unsigned* comparison.

CMPSWAP on the other bytes, for *signed* comparison.

Also, for signed comparison, the high byte operands must be in the opposite order from the others.

To compare for equality, still use `ltc`, but then use

```
fbeqlatch fbinv bspush
```

to push the equality latch with the proper assertion.

Example: unsigned multi-precision comparison

Assume three-byte unsigned numbers:

A = [L22, L21, L20] and B = [L19, L18, L17].

To implement

```
   if (A == B) {
```

do:

```
   addxz  L22, L22 ltc L19
   addxz  L21, L21 ltc L18 cmp
   addxz  L20, L20 ltc L17 cmp
   fbeqlatch fbinv bspush
```

Example: signed multi-precision comparison

Assume three-byte signed numbers:

A = [L22, L21, L20] and B = [L19, L18, L17].

To implement

    if (~~A~~ B < ~~B~~ A) {

do:

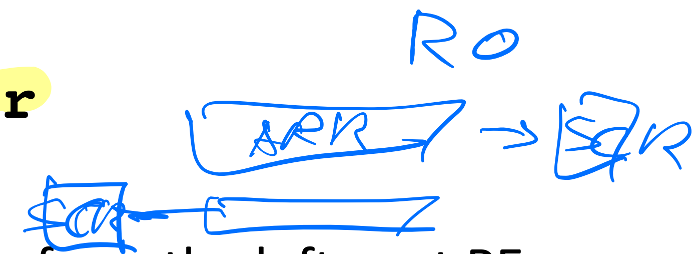
```
addxz   L19, L19 sltc L22
addxz   L21, L21 sltc L18 cmpswap
addxz   L20, L20 sltc L17 cmpswap bspush
```

TYPO

The controller’s scratch register: `arrtoscr`

ARRTOSCR “**ARR**ay **TO** controller **SCR**atch register” writes the result of the instruction into the controller’s scratch register as well as into the destination register.

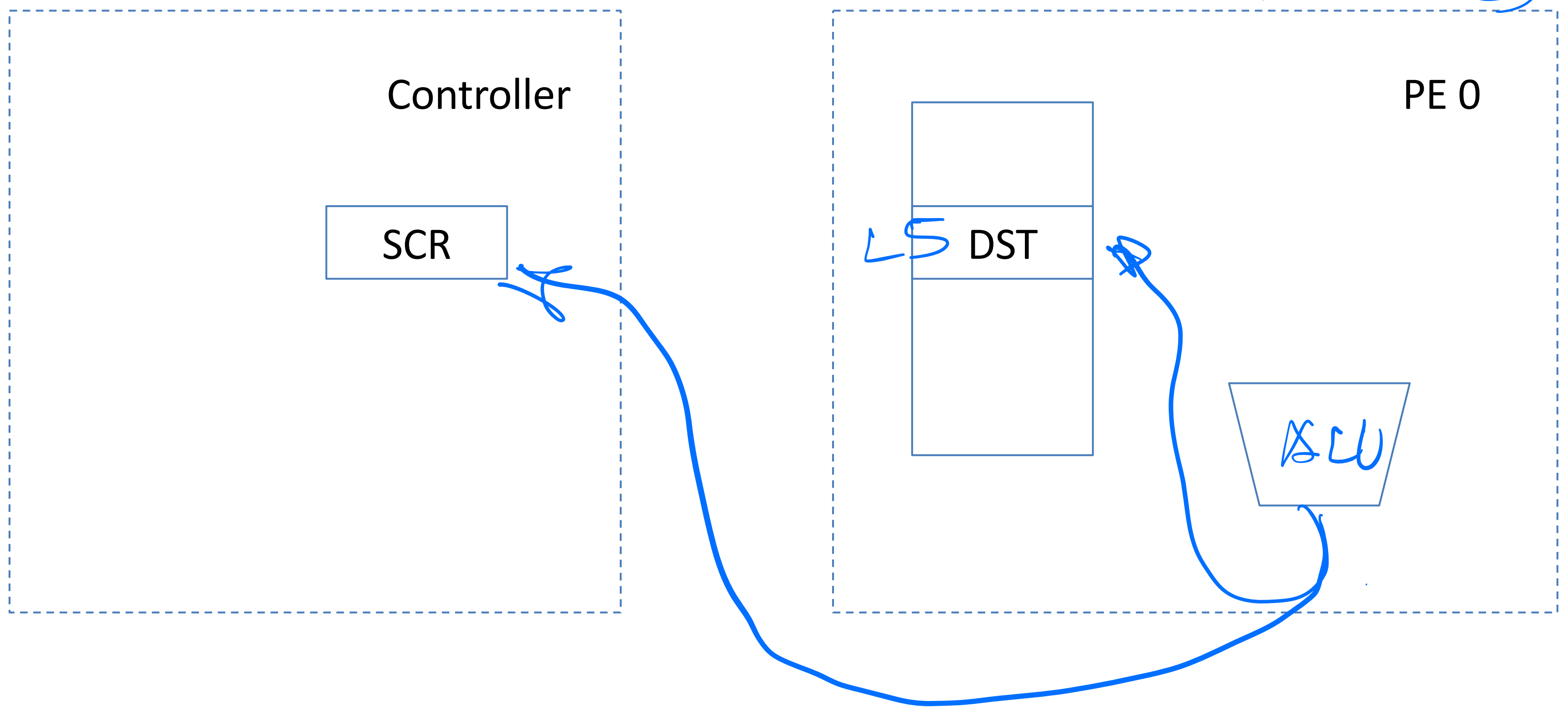
```
<instr>    Rdst, ... arrtoscr
```



- If **`Rdst`** is a left register, the value will come from the leftmost PE L0 in the array; if **`Rdst`** is a right register, the value will come from the rightmost PE in the array.
- If the outputting PE is masked, the scratch register (just like the destination register) **SHOULD NOT** be written (see Kasm manual) but **a bug in the simulator makes it write anyway**.

The controller's scratch register: `arrtoscr`

ADD L5 . . ARR TO SCR



The controller's scratch register: `scrtoarr`

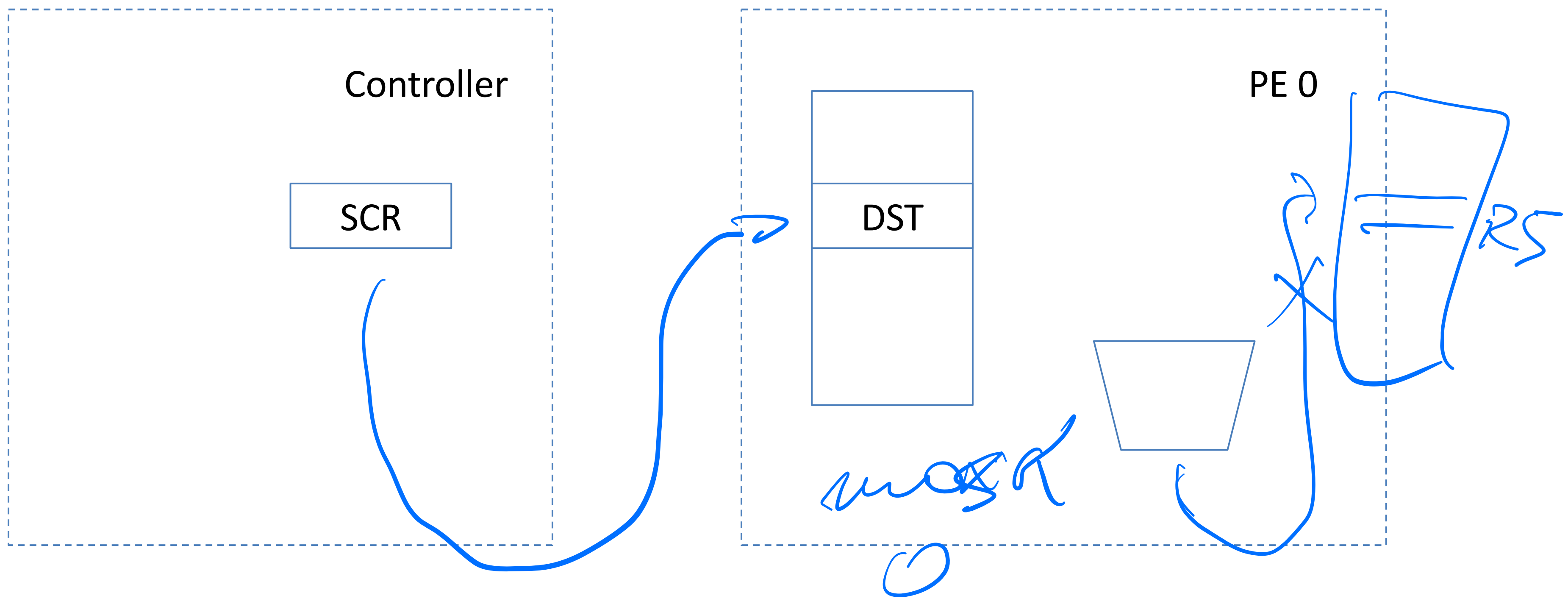
SCRTOARR "controller **SCR**atch register **TO ARR**ay" writes the content of the controller's scratch register into the destination register **`Rdst`**.

```
<instr>   Rdst, ... scrtoarr
```

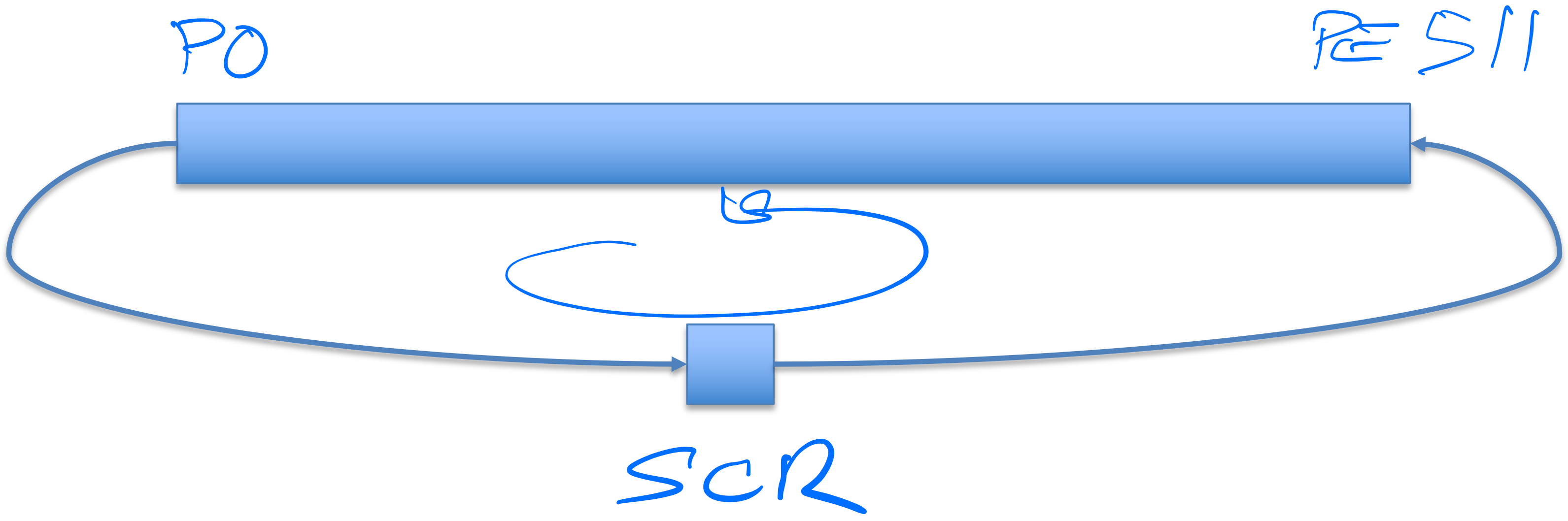
- If **`Rdst`** is a left register, the value will be written to the right register of the rightmost PE in the array; if **`Rdst`** is a right register, the value will be written to the left register of the leftmost PE in the array.
- If the receiving PE is masked, the destination register is written anyway.

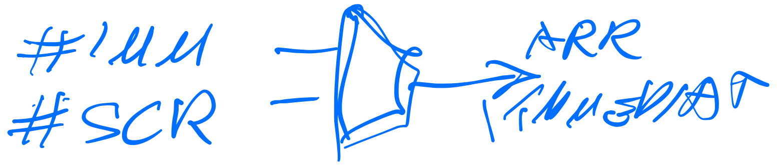
The controller's scratch register: `scrtoarr`

ADD R5, ... SCRTOARR



Loop through scratch





The scratch register: `scrtoimm`, `qtoscr`

SCRTOIMM “Controller **SCR**atch register **TO** array **IMM**ediate” replaces the array immediate field with the scratch register:

```
<instr using array imm>   scrtoimm
```

- Can be expressed as **`#scr`** whenever an array immediate is used.
- The scratch register cannot be written in the instruction immediately before.

BROADCAST

QTOSCR “Input **Q**ueue **TO** controller **SCR**atch register” writes a byte from the input queue to the scratch register:

```
<any array instr>   qtoscr
```

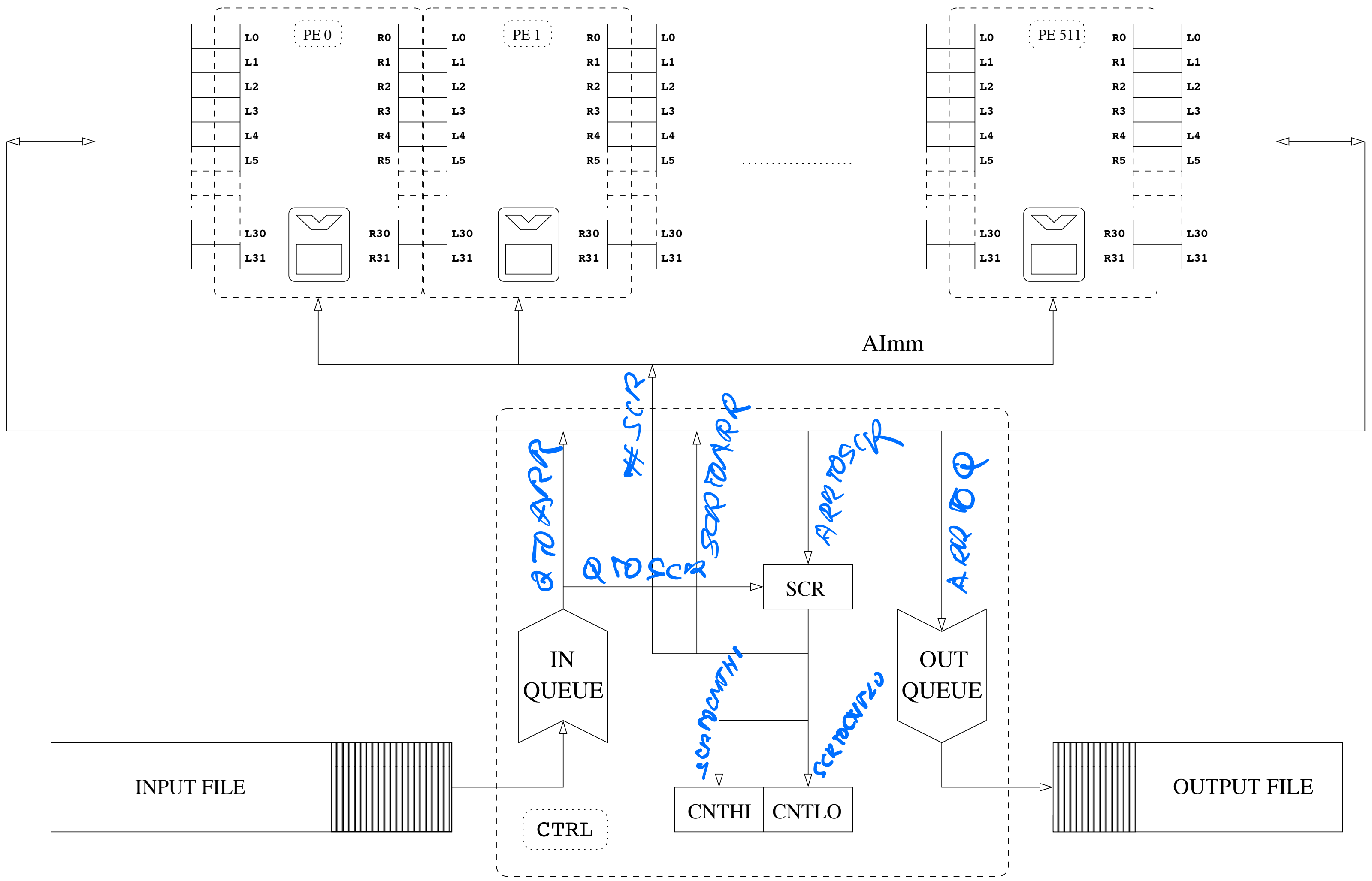
$O(1)$

One intervening instruction is needed between **`qtoscr`** and using the value in the scratch register.

(**SCRTOQ**) Does not exist! More on this later.

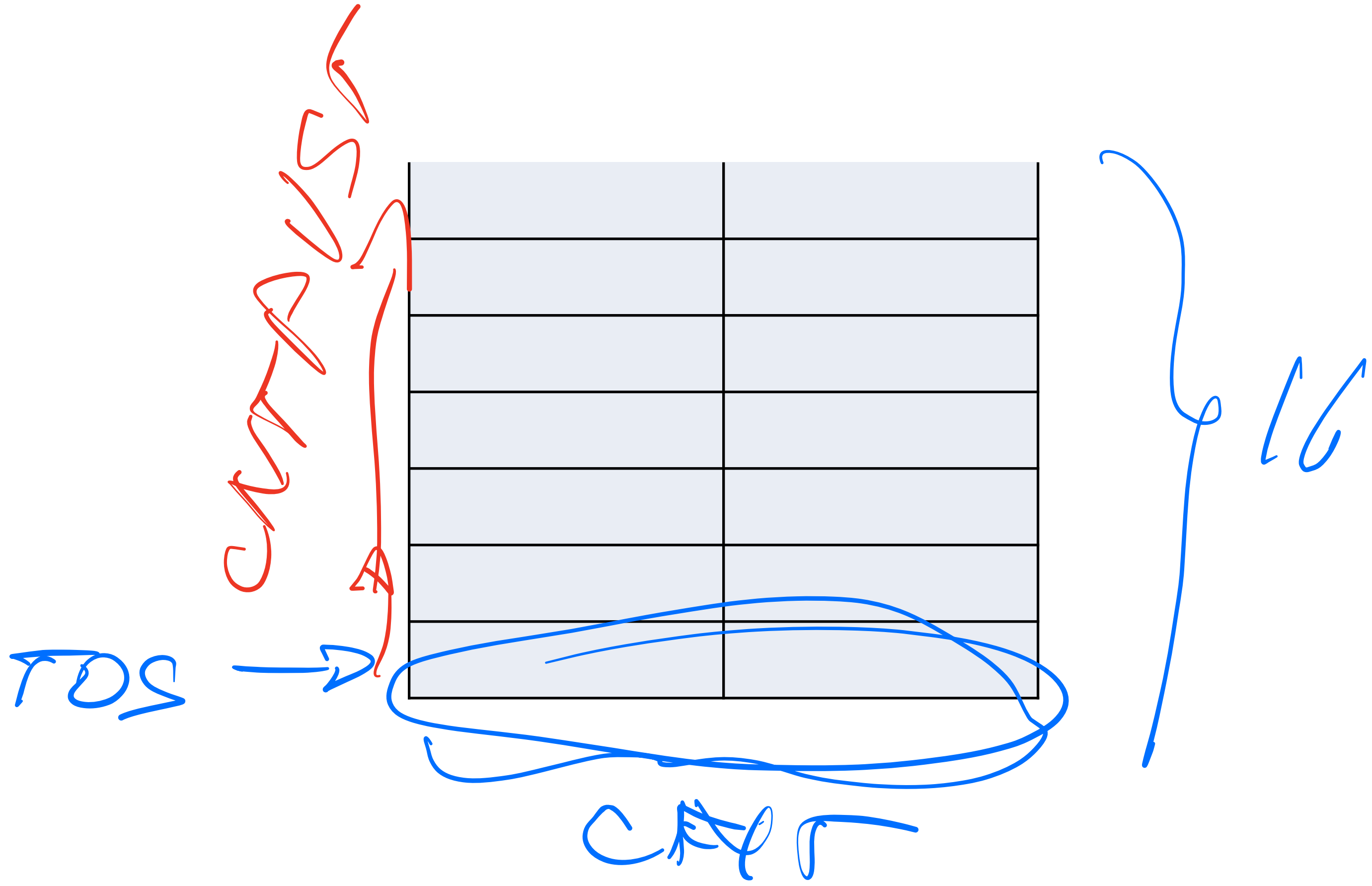
CONSTANT TIME

Controller's scratch register and loop counter



Loop counter stack

The controller's loop counter is a 16-deep *stack* of 16-bit counters:



Setting the loop counter at runtime

Two steps: first set the loop counter, then start the loop. To set the loop counter, there are two ways. The first one is the same as **`beginloop`**:

CNTPUSH "**C**ou**NT**er **PUSH**" pushes the controller's 16-bit immediate onto the counter stack.

```
<any array instr> cntpush CImm
```

≡ BEGINLOOP

Setting the loop counter at runtime

The second way uses an actual runtime value:

SCRTOCNTLO "**SCR**atch **TO** **C**ou**NT**er **LO**w byte" writes the scratch register into the low byte of the 16-bit loop counter (top of stack).

```
<any array instr>  scrtocntlo
```

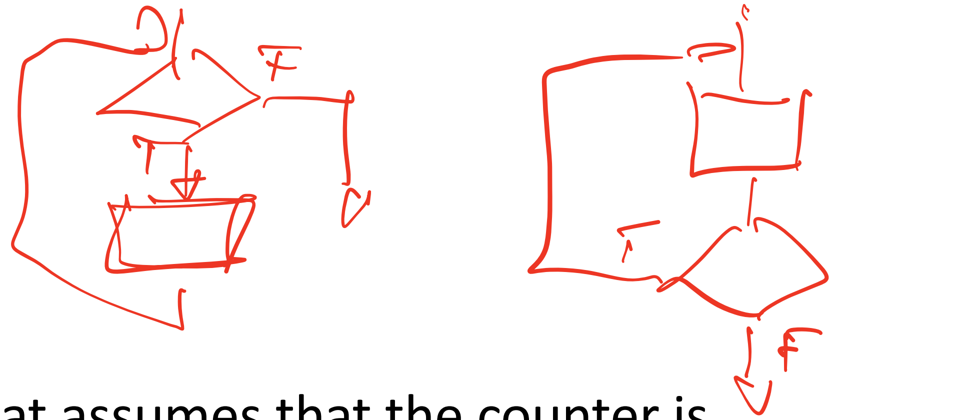
SCRTOCNTHI "**SCR**atch **TO** **C**ou**NT**er **HI**gh byte" writes the scratch register into the high byte of the 16-bit loop counter (top of stack).

```
<any array instr>  scrtocnthi
```

To preserve potential other values already on the stack, one must first make room on the stack by pushing, for instance, a dummy immediate value:

```
<any array instruction> cntpush 0
```

FOR = WHILE DO DO-WHILE



Starting the loop

To start the loop:

BEGINLOOPSCR Pseudo-instruction that assumes that the counter is already loaded and generates a label to match it with an **`endloop`** instruction.

```
<any array instr> beginloopscr
```

- Due to a design... feature, when loading the counter with a value *m*, the loop is actually executed *m* + 1 times.
- **`beginloopscr`** matches with an **`endloop`** instruction just like **`beginloop`**
- Can never execute a loop zero times, since it is a do-while loop.

Example of runtime loop

Assume that the number of times we want to loop is a 16-bit number registers [L24, L23] of the leftmost PE.

```
subxz    b1     L23, L23
subxz    mp     L24, L24
move            L23, L23, arrtoscr
cntpush 0
nop
scrtocntlo
move            L24, L24, arrtoscr
nop
nop
scrtocnthi
BEGINLOOPSCR
    move    L28, R28, qtoarr         ;   read row in
ENDLOOP
```

} 2 INSTR.

} 2 INSTR

POD

Example of runtime loop: better code

The code in the example can be optimized to better exploit Kestrel's instruction-level parallelism:

2 1 4 1 5

```
subxz    b1    L23, L23, arrtoscr
subxz    mp    L24, L24  cntpush 0
nop
scrtocntlo
move           L24, L24, arrtoscr
nop
nop
scrtocnthi

BEGINLOOPSCR
ENDLOOP  move    L28, R28, qtoarr   ; read row in
```

Nested loops

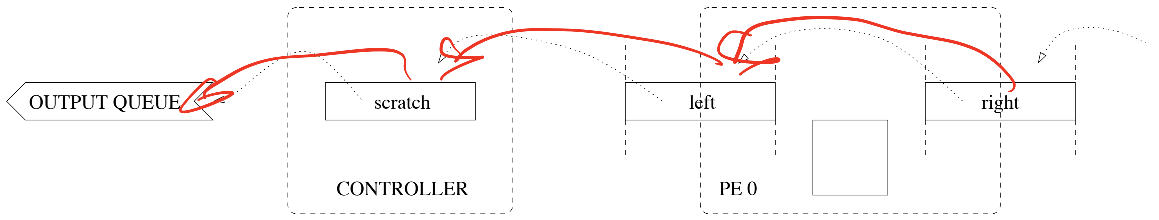
Loops can be nested to up to 16 levels deep. Make sure to keep that into account when using or nesting macros.

```
BEGINLOOP 10
  ...
  BEGINLOOP 20
    ...
    BEGINLOOP 30
      ...
    ENDLOOP
    ...
  ENDLOOP
  ...
ENDLOOP
```

Scratch register and I/O operations

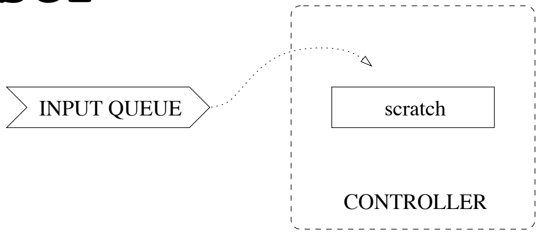
The scratch register is always written in an **`arrtoq`** operation:

```
move L0, R0, arrtoq
```



The scratch register can be written directly from the input queue:

```
<any instr>  qtoscr
```



Scratch register and I/O operations

There is no **`scrtoq`** instruction. Output from the scratch register has to go through one of the end PEs via the immediate field:

```
FORCE        move L0, #scr, arrtoq
```

or

~~`FORCEQOUT    move L0, #scr, arrtoq`~~

Local memory

READ Reads a byte from memory into the Memory Data Register (**`mdr`**).
The byte read is available to the next instruction as **`mdr`** operand (**`OpB`**).
Ex:

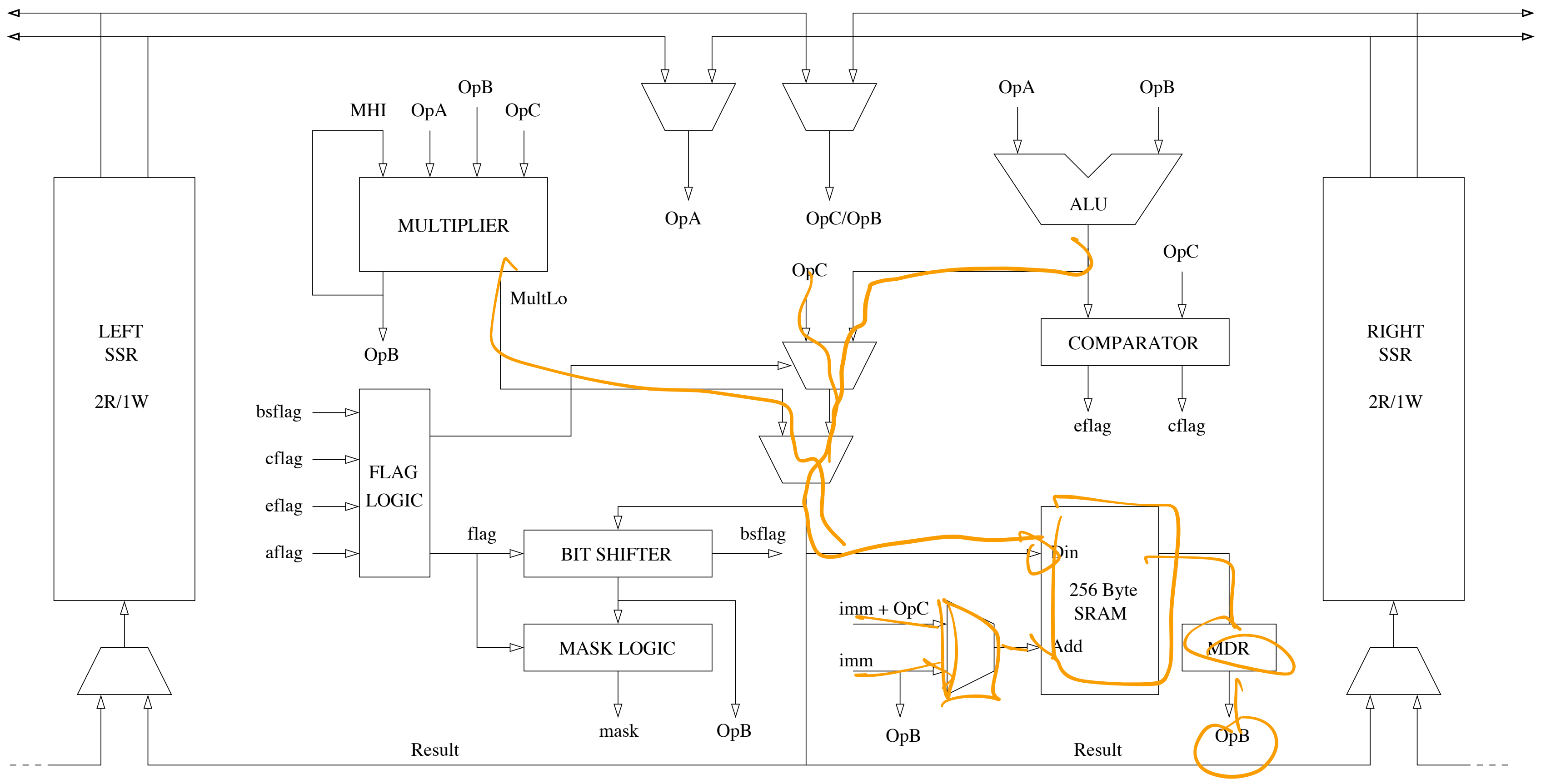
```
read(L3 + #120)
add         L10, L10, mdr, read(L3 + #121)
add   mp    L11, L11, mdr
```

WRITE Writes to memory the result of an operation. Example:

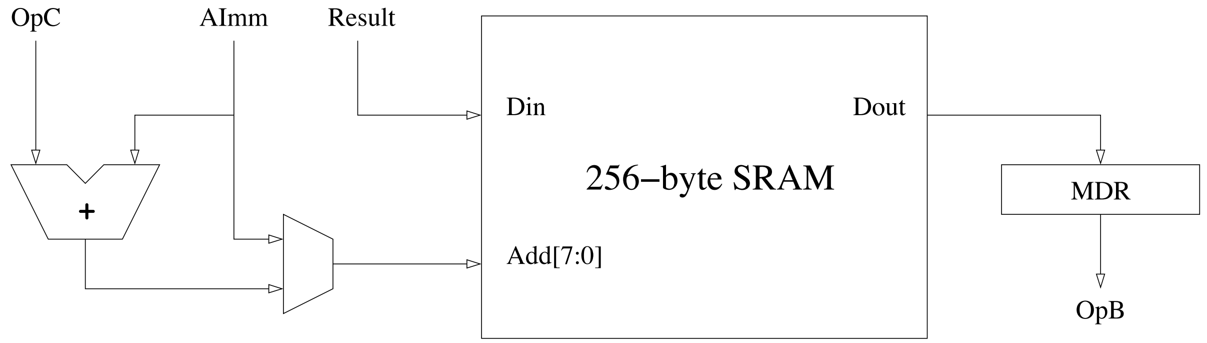
```
move        L10, L10, write(#42)
move        L11, L11, write(#43)
```

Memory reads and writes *cannot* appear in the same instruction.

Kestrel PE diagram



Addressing modes of local memory



Direct The address is the array 8-bit immediate.

`read(AImm)`

Indirect The address is the content of a register.

`read(Reg)`

Base The address is the sum of a register and the immediate.

`read(Reg + AImm)`

The array immediate is always used (so `indirect' is really `base'). Therefore even when only a register is specified, no array immediate can be used anyway. NOTE that if you do, the results are undefined, and the assembler will not warn you! Example: ***can't do*** `MOVE L0, #3, write(L1)`