EE382A – Spring 2025

Chapter 3: Kestrel’s Instruction Set Architecture (2 of 5)

Kestrel's ISA: second part

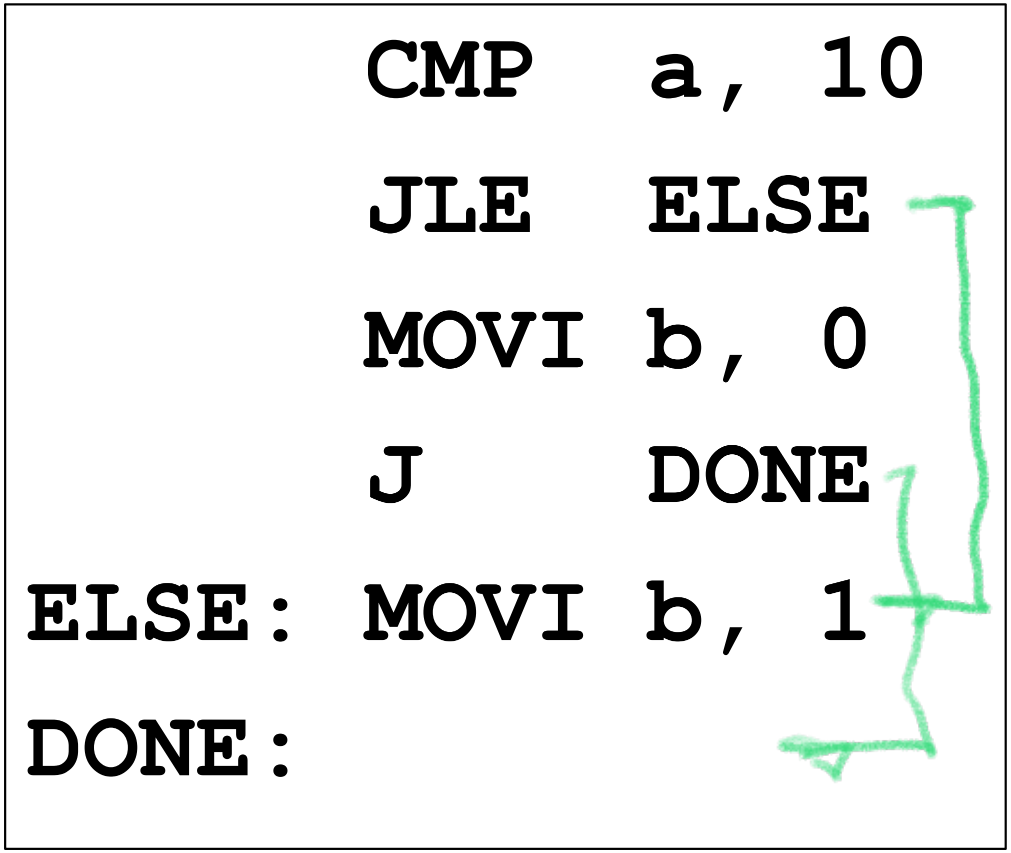
- Conditional execution: the active set
- Comparison instructions
- Selection instructions
- Breakpoints
- Assembler directives: **`define`**, **`include`**, and macros

Conditional execution: *serial* code

Serial source code

```
if (a > 10)
{
  b = 0;
} else {
  b = 1;
}
```

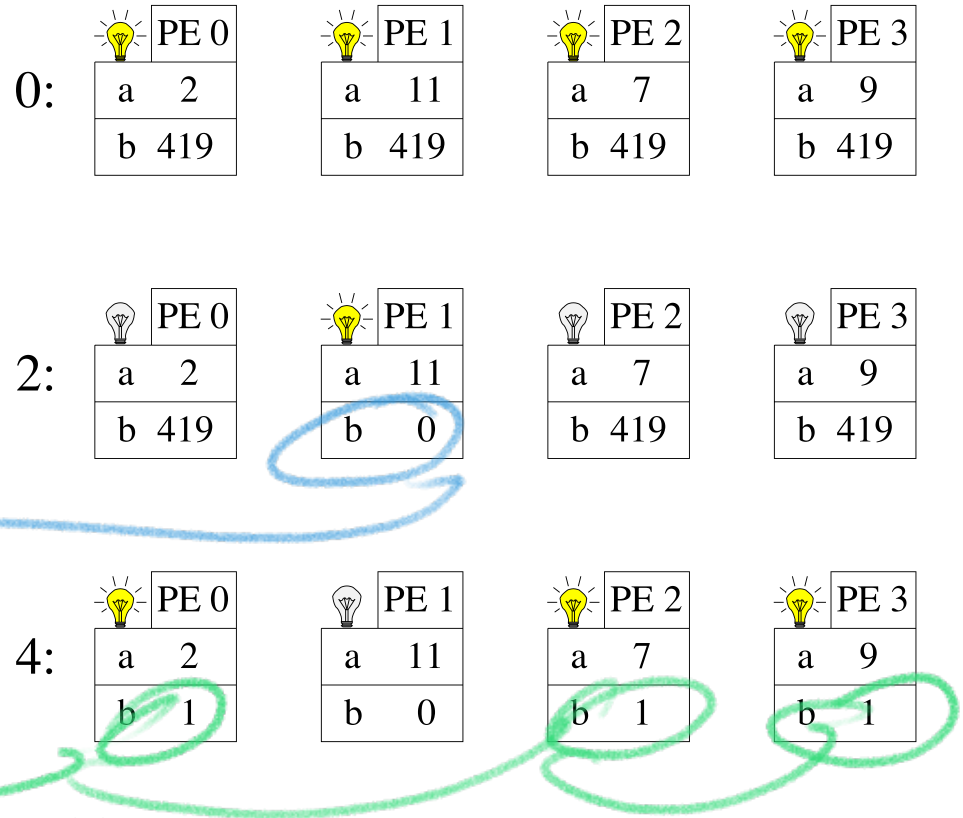
Serial assembly code



```
      CMP  a, 10
      JLE  ELSE
      MOVI b, 0
      J    DONE
ELSE: MOVI b, 1
DONE:
```

Conditional execution in SIMD: *active set*

```
plural  int   a, b;

    . . .

1:  if(a > 10)
2:     b = 0;
3:  else
4:     b = 1;

    . . .
```



		PE 0	PE 1	PE 2	PE 3
0:		●	●	●	●
	a	2	11	7	9
	b	419	419	419	419
2:		○	●	○	○
	a	2	11	7	9
	b	419	0	419	419
4:		●	○	●	●
	a	2	11	7	9
	b	1	0	1	1

In SIMD there are *singular* and *plural* variables.

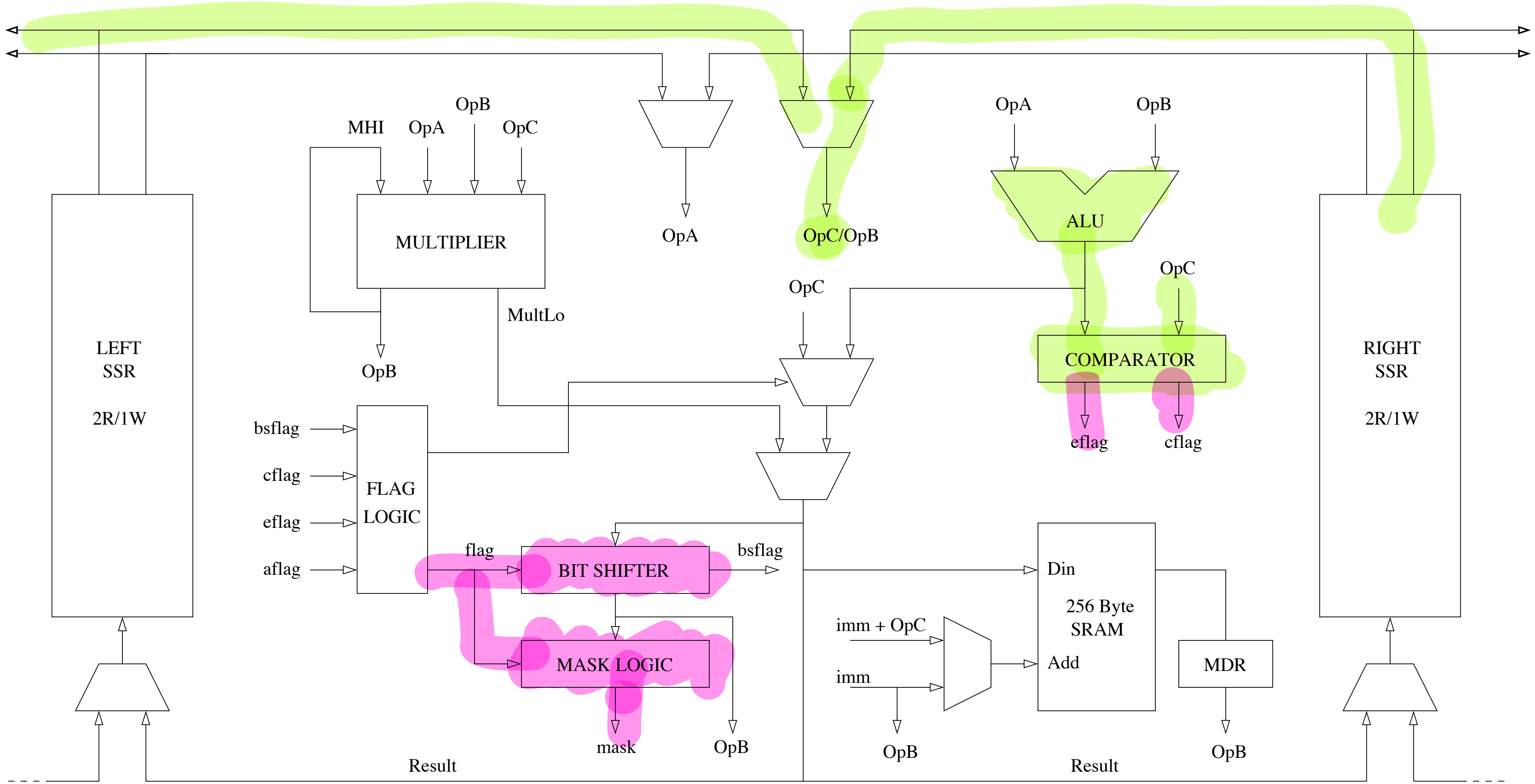
Singular variables have the same name and *the same value* in all PEs.

Plural variables have the same name but can have different values in different PEs.

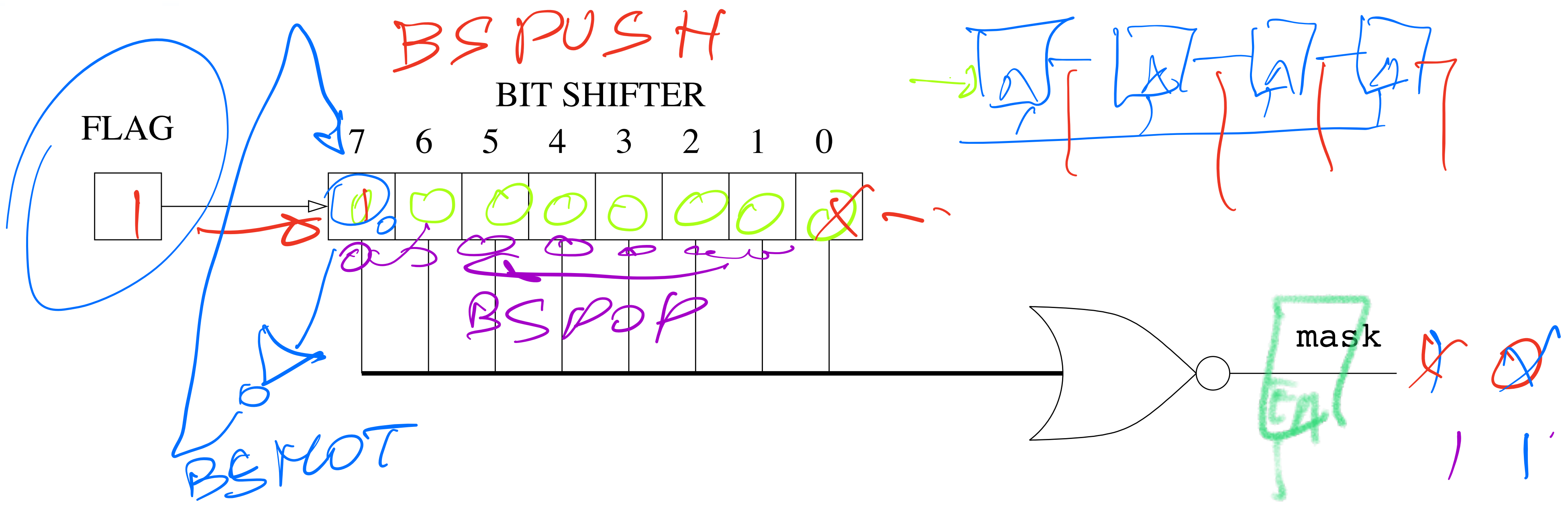
A test on a plural variable defines the *active set*.

More sophisticated SIMD machines can test and jump on a singular variable too.

Active set in Kestrel: the comparator



Active set in Kestrel: the bit shifter



`mask` = 1 means PE is ON, **`mask`** = 0 means PE does not *write* (OFF).

`mask` is NOR of all bits in the bit shifter. Any bit at 1 turns PE OFF.

Use bit shifter as a stack to implement nested conditionals (push the **`flag`** register into msb of BS).

Must set the **`flag`** register based on some condition.

Example of equality comparison

Kestrel assembly code

INVERTED LOGIC: TRUE = 0

```
addxz L0, L0 equalc L15 bspush
          7            20        1

    move L25, #1   IDLE

bsnot

    move L25, #2   ✓

bspop
```

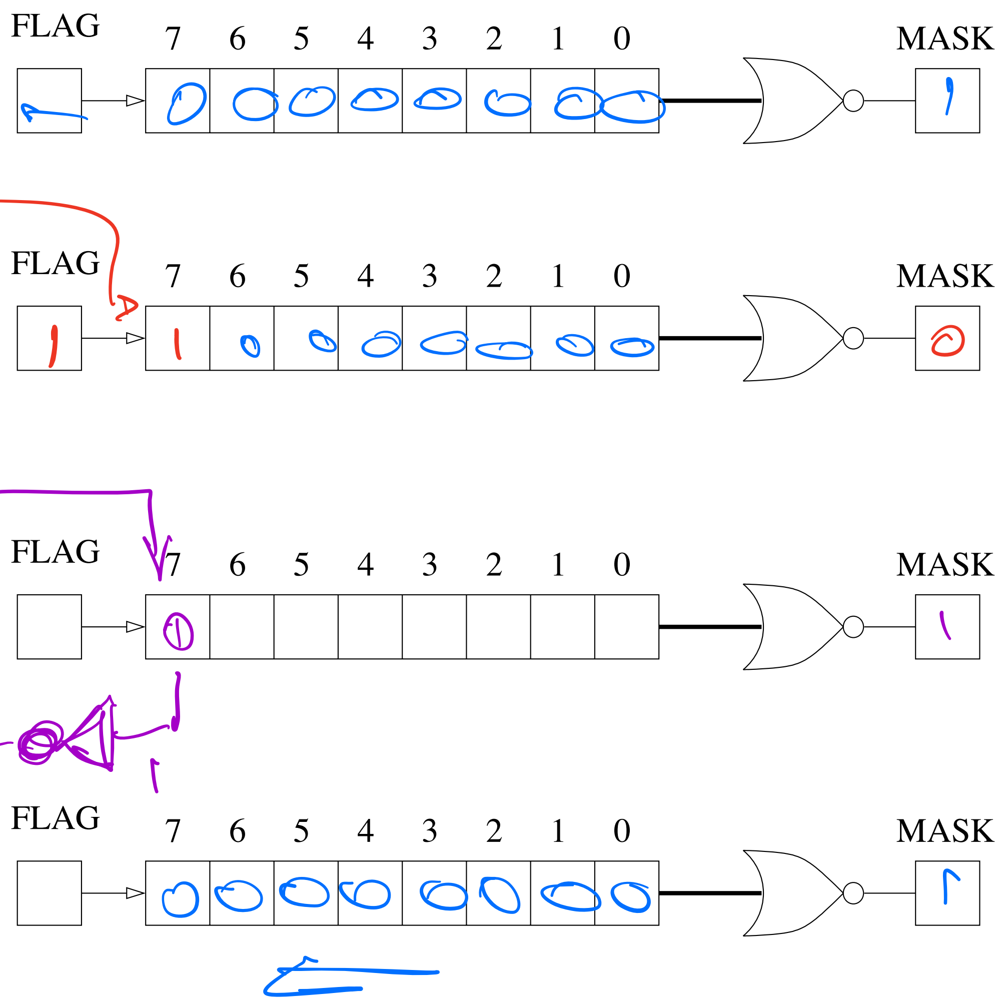
"Equivalent" serial code

```
if (L0 == L15) {

  L25 = 1;

} else {

  L25 = 2;

}
```

Example of equality comparison

L0	L15	L19	L20	L25
7	20			~~0~~ 2

```
addxz L0, L0 equalc L15 bspush
      7              20

    ~~move L25, #1~~

bsnot

    move L25, #2  ✓

bspop
```



Comparison instructions

EQUALC Sets FLAG = 0 if the result of an ALU operation is equal to operand **`OpC`.**

```
<ALU op>  equalc  <OpC>
```

LTC Sets FLAG = 0 if the *unsigned* result of an ALU operation is less than the *unsigned* operand **`OpC`**.

```
<ALU op>  ltc  <OpC>
```

SLTC Sets FLAG = 0 if the *signed* result of an ALU operation is less than the *signed* operand **`OpC`**.

```
<ALU op>  sltc  <OpC>
```

Bit-shifter instructions for conditionals

BSPUSH Pushes the FLAG onto the bit-shifter's msb. The PE mask is evaluated. This is equivalent to an ``IF'' statement.

```
<comparison instr.>  bspush
```

BSNOT Inverts the most significant bit of the bit-shifter. The PE mask is evaluated. This is equivalent to an ``ELSE'' statement.

```
bsnot
```

BSPOP Pops the most significant bit off the bit shifter, shifting a zero into the lsb. The PE mask is evaluated. This is equivalent to an ``ENDIF.’’

```
bspop
```

BSCLEARM Sets all bits in the BS to zero and evaluates the mask (to 1). Used to initialize the conditional execution in all PEs (all ON).

```
bsclearm
```

NOTE that all the above instructions occur in **all PEs** regardless of the mask.

Example of nested conditionals (AND)

Kestrel assembly code

```
addxz L0, L0 equalc L15 bspush

  addxz L20, L20 ltc L19 bspush

    move L25, #1
  bspop
bsnot

    move L25, #2

bspop
```

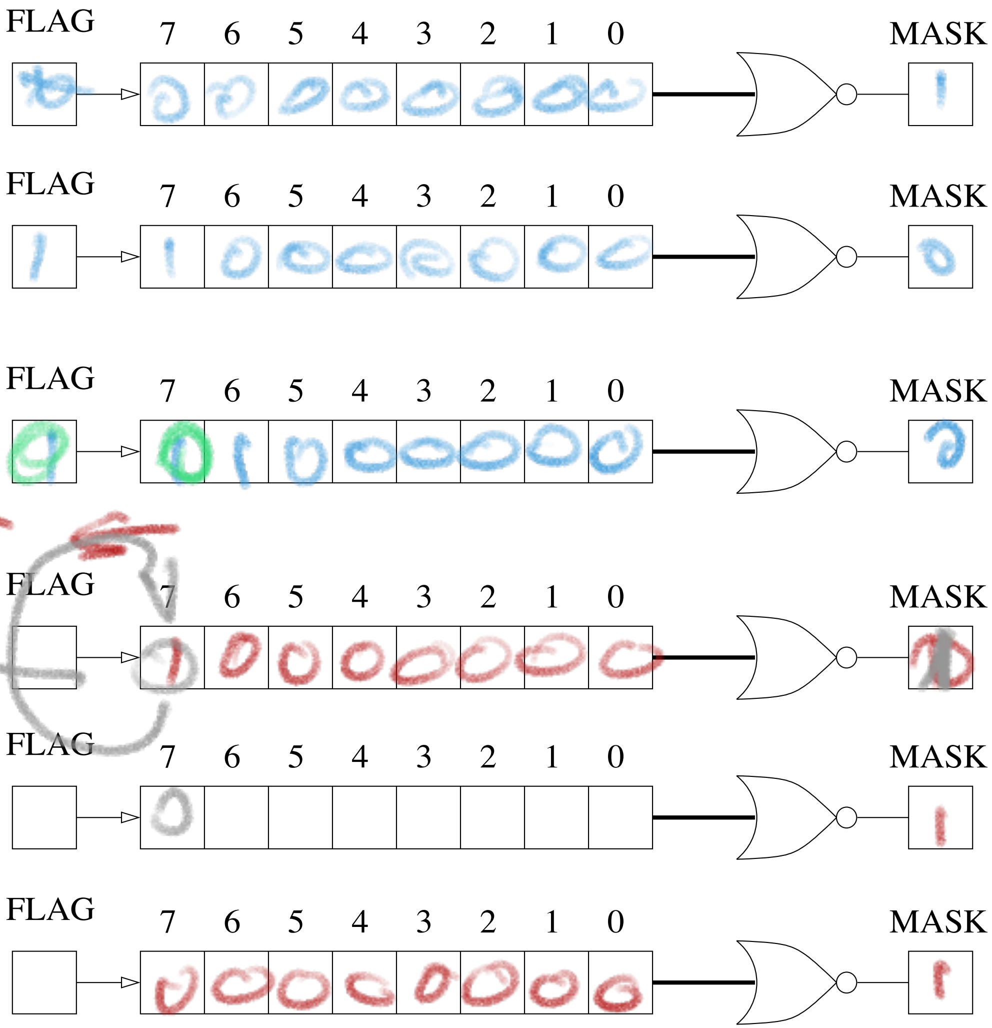
"Equivalent" serial code

```
if (L0 == L15) {

  if (L20 < L19) {

    L25 = 1;
  }
} else {

  L25 = 2;

}
```

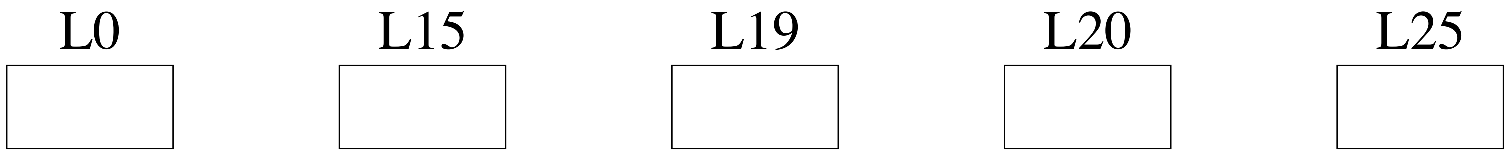
Example of nested conditionals (AND)

L0	L15	L19	L20	L25
7	20	12	11	~~34~~ 2

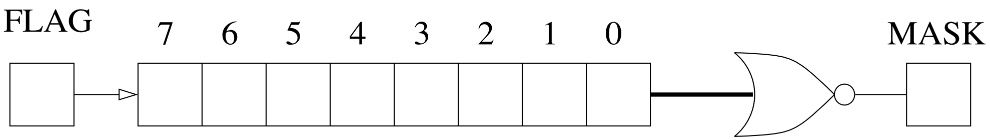
```
addxz L0, L0 equalc L15 bspush
          7          20     1
  addxz L20, L20 ltc L19 bspush
          11         12     0
    move L25, #1
  bspop
bsnot
    move L25, #2
bspop
```

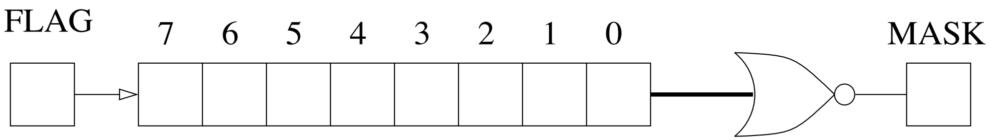


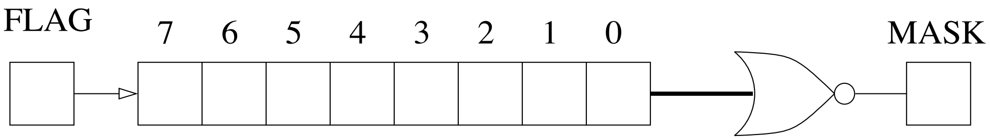
Example of faster nested conditionals

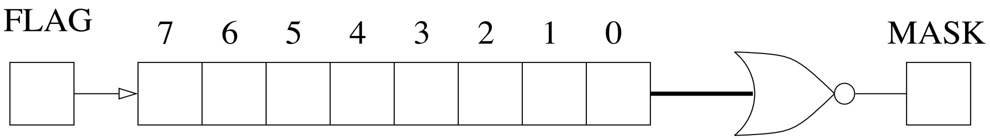


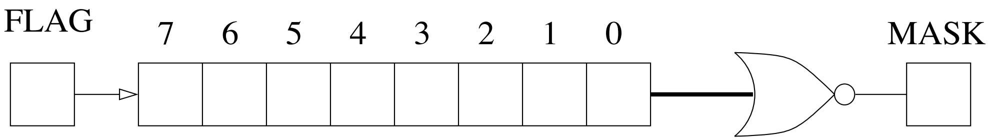
```
addxz L0, L0 equalc L15 bspush

  addxz L20, L20 ltc L19 bspush

    move L25, #1
bspopnot


    move L25, #2


bspop
```











Comparison *less-than-or-equal*

Use **`ltc`** first, and then combine with the **equal flag**.

FBEQLATCH Sets the flag to the comparator's equality latch. Note that, unlike **`equalc`**, the equality latch is 1 when the two operands are equal (use **`fbinv`** with bit shifter).

```
fbeqlatch
```

FBINV Inverts the flag value from that defined by the associated instruction (pseudo).

```
<any flag-setting inst.>  fbinv
```

BSAND ANDs the flag bit with the msb of the bit shifter, and re-evaluates the mask. Note that, considering the encoding, the TRUE value is 0, so this operation is actually a logical OR for the mask.

```
<any flag-setting inst.>  bsand
```

Example of less-than-or-equal

Kestrel assembly code

10 ≤ 10

```
addxz L0, L0 ltc L15 bspush

nop fbeqlatch fbinv bsand

    move L25, #1

bsnot

    move L25, #2

bspop
```

1

0 → 0

✓

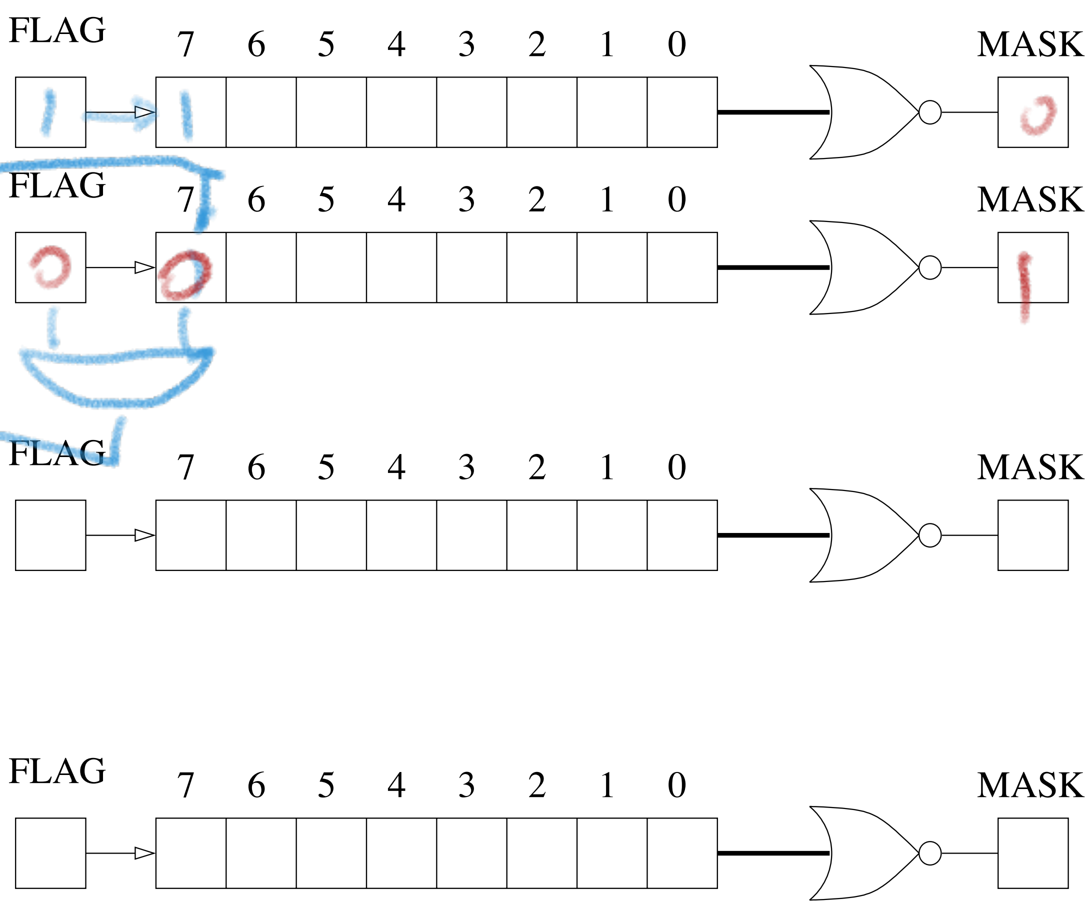
"Equivalent" serial code

```
if (L0 <= L15) {
  L25 = 1;
} else {
  L25 = 2;
}
```

Example of less-than-or-equal

L0	L15	L19	L20	L25
10	10	-	-	-

```
addxz L0, L0 ltc L15 bspush
       10      <  10      1

nop fbeqlatch fbinv bsand
         1        0

    move L25, #1

bsnot

    move L25, #2

bspop
```



Example of less-than-or-equal

DE MORGAN'S

ltc <	fbeqlatch fbinv =	<=	bsand
T 0	T 0	T 0	T 0
T 0	F 1	T 0	T 0
F 1	T 0	T 0	T 0
F 1	F 1	F 1	F 1

Selection instructions: unsigned

MAXC maximize with C: **`Dst`** gets the maximum of the unsigned operands **`OpA`** and **`OpC`**.

```
maxc Dst, OpA, OpC
```

MINC minimize with C: **`Dst`** gets the minimum of the unsigned operands **`OpA`** and **`OpC`**.

```
minc Dst, OpA, OpC
```

Selection instructions with ALU operation

MAXC maximize with C: **`Dst`** gets the maximum of the unsigned ALU result of operation **`<op>`** and of the unsigned operand **`OpC`**.

```
<op> Dst, OpA, OpB maxc OpC
```

MINC minimize with C: **`Dst`** gets the minimum of the unsigned ALU result of operation **`<op>`** and of the unsigned operand **`OpC`**.

```
<op> Dst, OpA, OpB minc OpC
```

Selection instruction based on the flag

SELECTC **`Dst`** gets **`OpC`** if **`flag`** = 0, the result otherwise. Can also be used without specifying an ALU instruction.

One must specify the flag used, and any flag-setting instruction can be used.

```
<op> Dst, OpA, OpB selectc OpC fb*
selectc Dst, [OpA | OpB], OpC fb*
```

Note that you have to specify the flag used. To perform

```
L10 = (L0 < L1) ? L2 : L3
```

This will NOT work (assembler error):

```
addxz L0, L0 ltc L1   ; flag gets set
selectc L10, L2, L3   ; use flag previously set
```

You have to do this:

```
addxz L0, L0 ltc L1   ; minlatch set to 0 if L0 < L1
; L10 = (minlatch == 0) ? L2 : L3 (L2 is OpC)
selectc L10, L3, L2 fbminlatch
```

Conditional code and **FORCE**

Conditional execution can be overridden by setting a special bit in the instruction, the **FORCE** bit. When this bit is set, all PEs are active regardless of their mask.

This mode (**FORCE** bit set) is the default for the assembler.

Conditional code (**FORCE** bit not set) is only generated by the assembler for the code in between the two directives:

BEGINCOND

and

ENDCOND

One can still force all PEs to execute a specific instruction regardless of their mask value even in conditional code by pre-pending the directive **FORCE** that will set the **FORCE** bit for the instruction.

Code writing suggestions

Apart from special cases, it is good practice to assume the opposite of the assembler's default, and make *all code conditional*.

Always enclose all program instructions in between **`BEGINCOND/ENDCOND`** pair.

Use **`FORCE`** as little as possible, only when needed.

When writing a macro, always assume that a it will be called in a conditional piece of code.

Use **`bsclearm`** as the first instruction in every program: the simulator automatically resets all mask registers (all active), but the (future) board may not.

A standard program template

```
;=========================================
; program: programName.kasm
; <whatever other comments>
;=========================================
start:
bsclearm
BEGINCOND
   [...]
   <entire program in conditional space>
   [...]
FORCE <instruction>   ; this instruction always executes
   [...]
ENDCOND
end:
;=========================================
```

Debugger: breakpoints

- Insert breakpoints in the code
- The debugger's **`run`** command stops at breakpoints
- A list of all breakpoints in the program is shown at debugger startup
- The last breakpoint is the program end
- Useful to evaluate number of instructions (and therefore clock cycles) for portions of the code
- Bug in the debugger: do not set/unset breakpoints using the **`breakpoint`** menu - use the **`breakpoint`** instruction in the code instead

Breakpoints

- NOTE: dump doesn't work in the debugger
- TRICK to have breakpoints triggered dynamically:
  - use jumpwor to skip a breakpoint a number of times

Breakpoints example

```
;***********************************
; Program: kex01bp.kasm
; First Kestrel program example with breakpoints
;***********************************
start:    ; program execution starts here
breakpoint
addzz    L0
breakpoint
add      L0, L0, #3
;***********************************

$ kestrel -debug kex01bp.ko kin kout 4
Kestrel Run Time Environment (compiled 05/08/99_17:57:43)
Copyright (c) 1998 Regents of the University of California

kestrel rte: Starting the Kestrel Serial Simulator.
kestrel rte: kex01.ko: breakpoint 1 detected at instruction 1
kestrel rte: kex01.ko: breakpoint 2 detected at instruction 3
kestrel rte: kex01.ko: end of program detected at instruction 5
kestrel rte: Starting the Kestrel Debugger (kdb)
```

Debugger: the mask latch

In the debugger, use **`masklatch`** in the **`range`** menu to inspect the mask value of all PEs. ONE means the PE is ON. Example:

```
kdb> masklatch
```

[bunch of messages – ignore them]

```
         maskLatch values from 0 to 511 are:
   0: 1    1: 0    2: 1    3: 0
   4: 1    5: 0    6: 1    7: 0
   8: 1    9: 0   10: 1   11: 0
...
```

Debugger: the bslatch

In the debugger, use **bslatch** in the **range** menu to inspect the bit shifter value of all PEs. Example:

```
kdb> bslatch
```

[bunch of messages – ignore them]

```
          bsLatch values from 0 to 3 are:
  0:    0    1:    0    2:    0    3:    0
```

DEFINE assembler directive

The **DEFINE** assembler directive is used to define symbols, as in

```
DEFINE     NPES    512
```

A defined **symbol** can be referenced as **$symbol**, as in

```
beginloop $NPES
```

Symbols are case-insensitive and may only include letters and underscores.

DEFINE must occur on a line by itself. **newkasm** does not support static expression evaluation, but resolves nested definitions, as in

```
DEFINE     NPIXELS   $NPES
```

Definitions and labels have dynamic scope with respect to macros: if **X** is defined in macro **M1**, and **M1** calls macro **M2**, then **X** is defined in macro **M2.**

Symbols are not limited to defining numbers, as in

```
DEFINE     LREG   L8
```

INCLUDE assembler directive

The **INCLUDE** assembler directive is used to include a source file, as in

```
INCLUDE  kConvolMacro.kasm
```

Macros

To define, use **MACRODEF** and **MACROEND**, as in:

```
MACRODEF COPY_NEIGHBOR_PIXELS()
      move    R6,  L7
      move    L8,  R7
      move    R4,  L5
      move    L10, R9
 MACROEND
```

To call, use the name, as in:

```
COPY_NEIGHBOR_PIXELS()
```

Macros can call other macros. Assembler errors of the form

`x.kasm:5:x.kasm:20` indicate that an error occurred in a macro's line 5, and that the macro was called from the main program at line 20.

Macros with register parameters

Macros can take registers or immediate values as parameters.

Parameters are treated as symbols and follow the same rules and are subject to the same max symbol count per program. Example:

```
MACRODEF ROW_SUM(SumH, SumL)
   add           $SumL, L0, L1
   addzz  mp     $SumH
   add           $SumL, $SumL, L2
   addxz  mp     $SumH, $SumH`
MACROEND
```

To call:

```
ROW_SUM(SumH, SumL)
```

Macros with immediate parameters

As controller constants:

```
MACRODEF output(leftR, rightR, PEs)
    BEGINLOOP $PEs
    ...
```

Called as:

```
output(L0, R0, 9)
```

As array immediates:

```
MACRODEF sortTwo(PE)
    move L29, #$PE
    ...
```

Called as:

```
sortTwo(1)
```