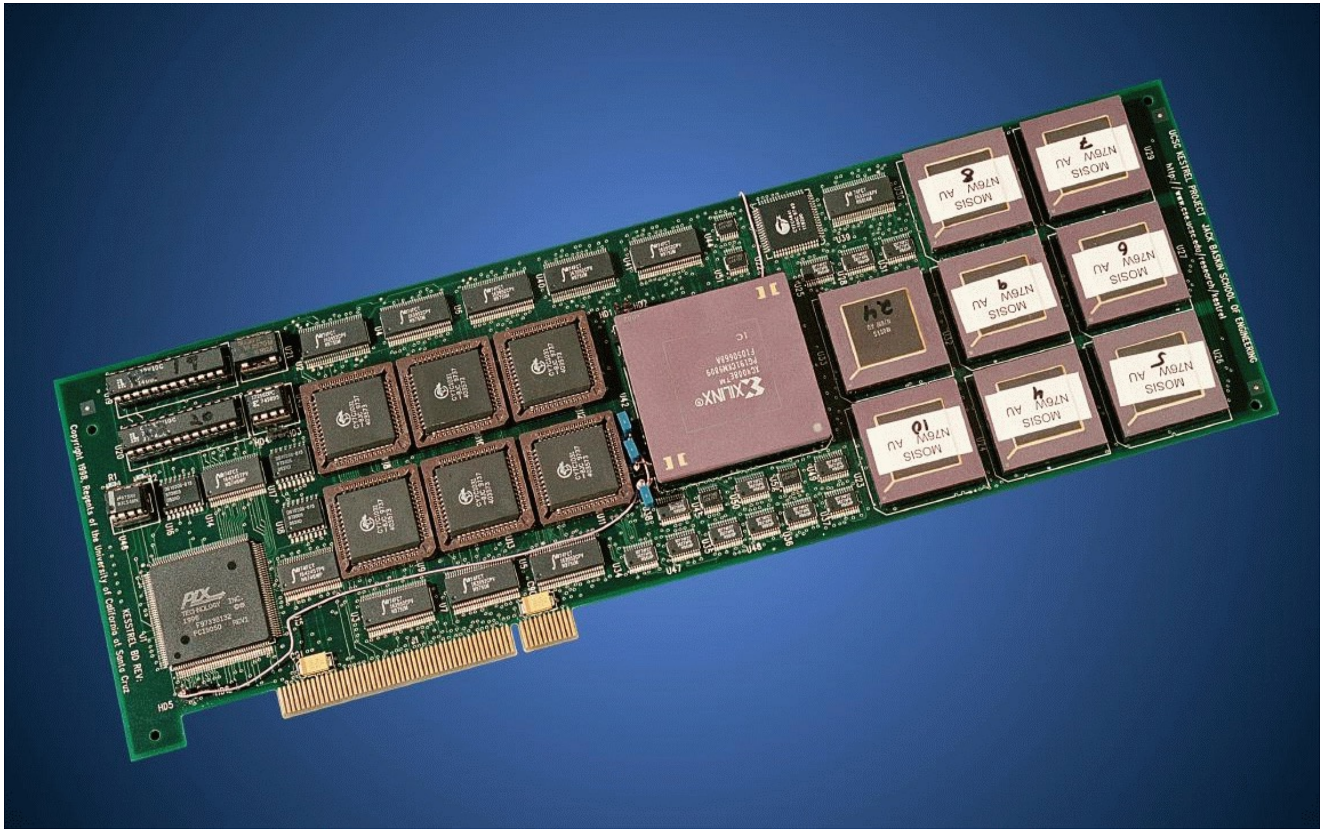
EE382A – Spring 2025

Chapter 2:
Kestrel's Instruction Set Architecture (1 of 5)

Kestrel's ISA, first part

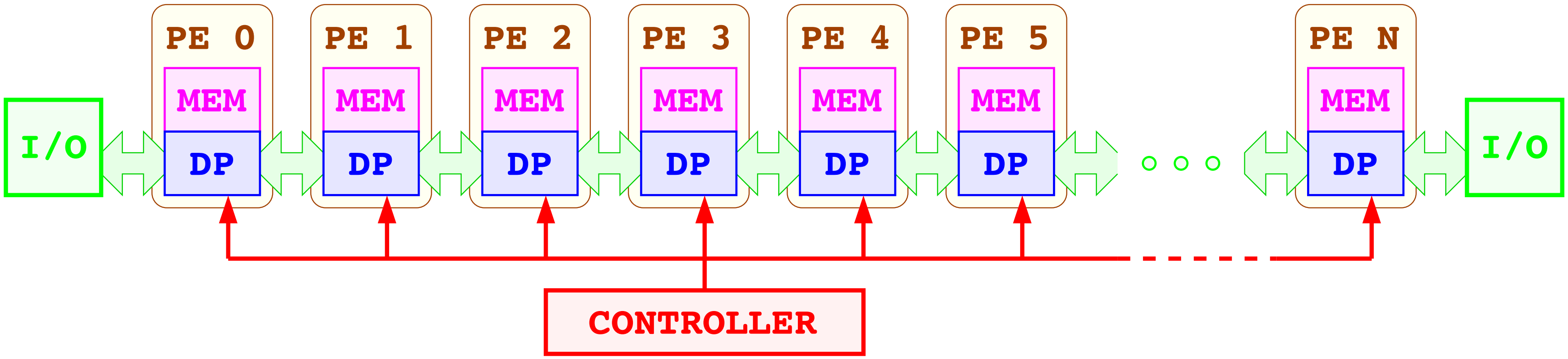
- Overview of Kestrel's ISA
- First program: compiling, running, debugging
- Second program: PE numbering, loops
- Third program: move, I/O
- Kestrel's streaming I/O system
- Programming assignment 1: PE count, PE alternate

UCSC Kestrel

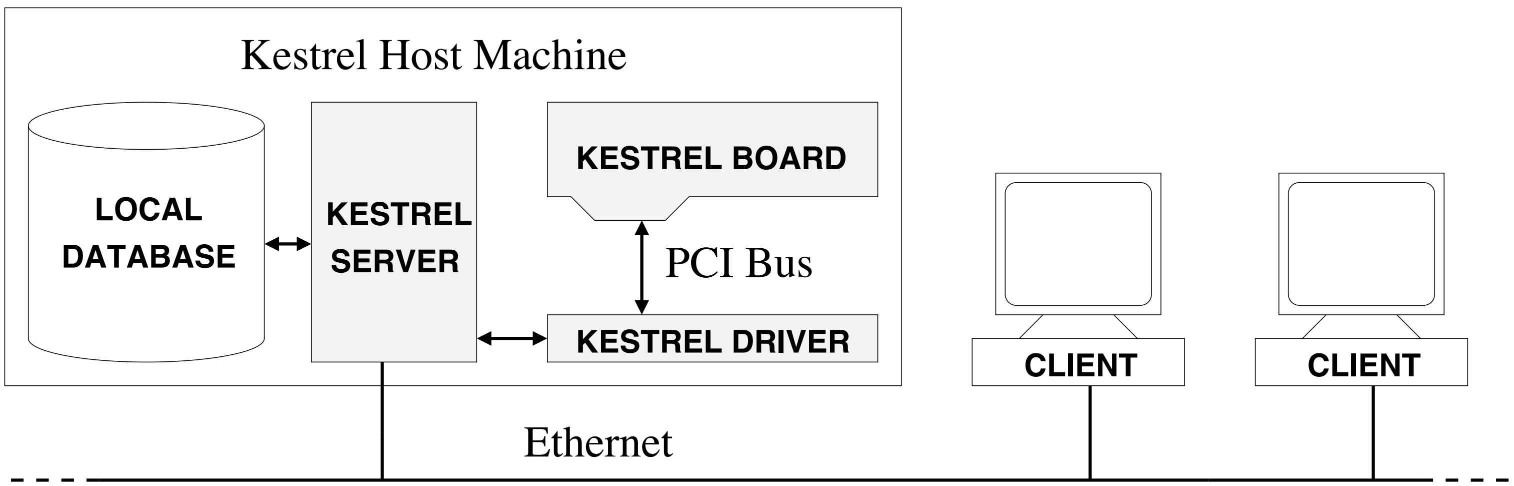


- 512 8-bit PEs, 256 bytes per PE, 20 MHz
- Linear SIMD array with wrap-around
- About 25 8-bit Gops peak (MAA = 2.5 ops)

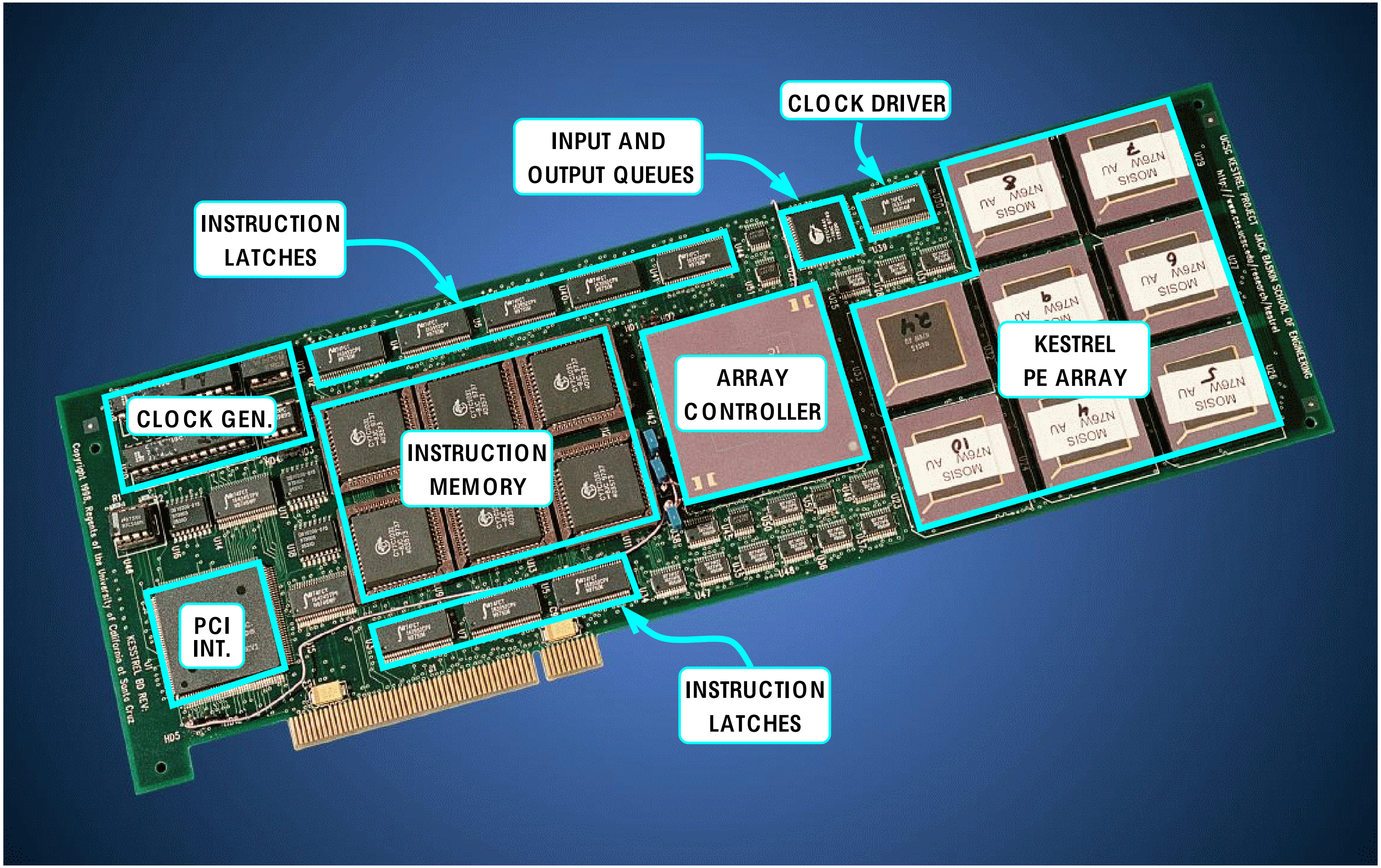
UCSC Kestrel: Linear SIMD array



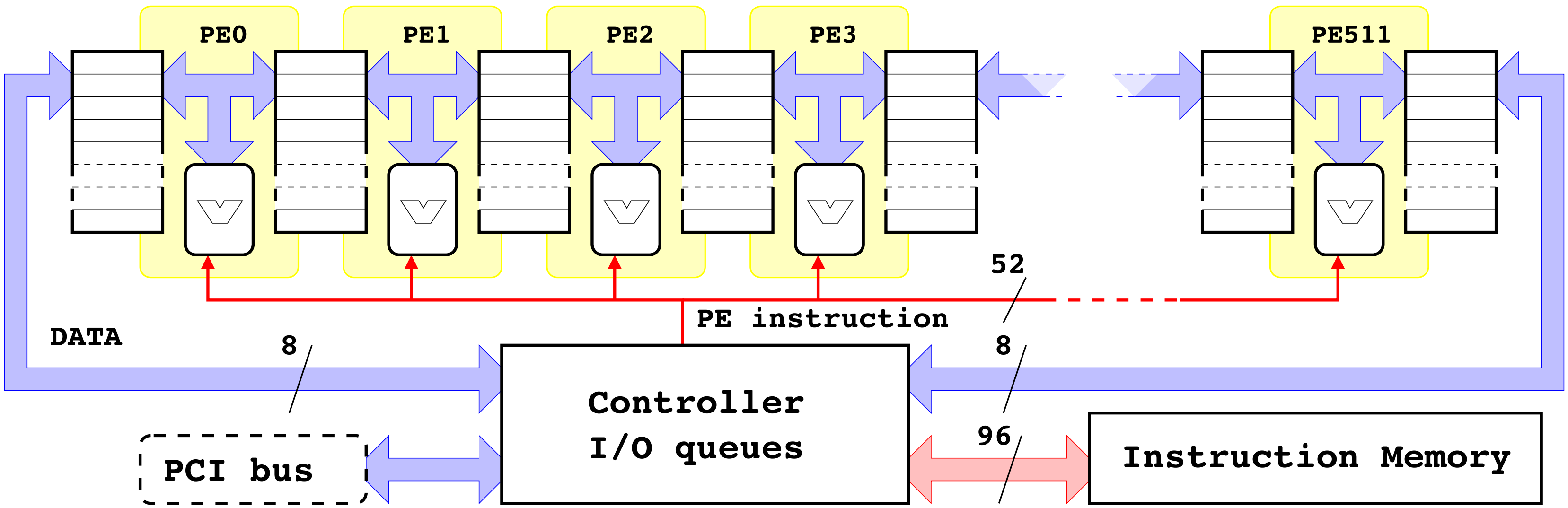
UCSC Kestrel: host and clients



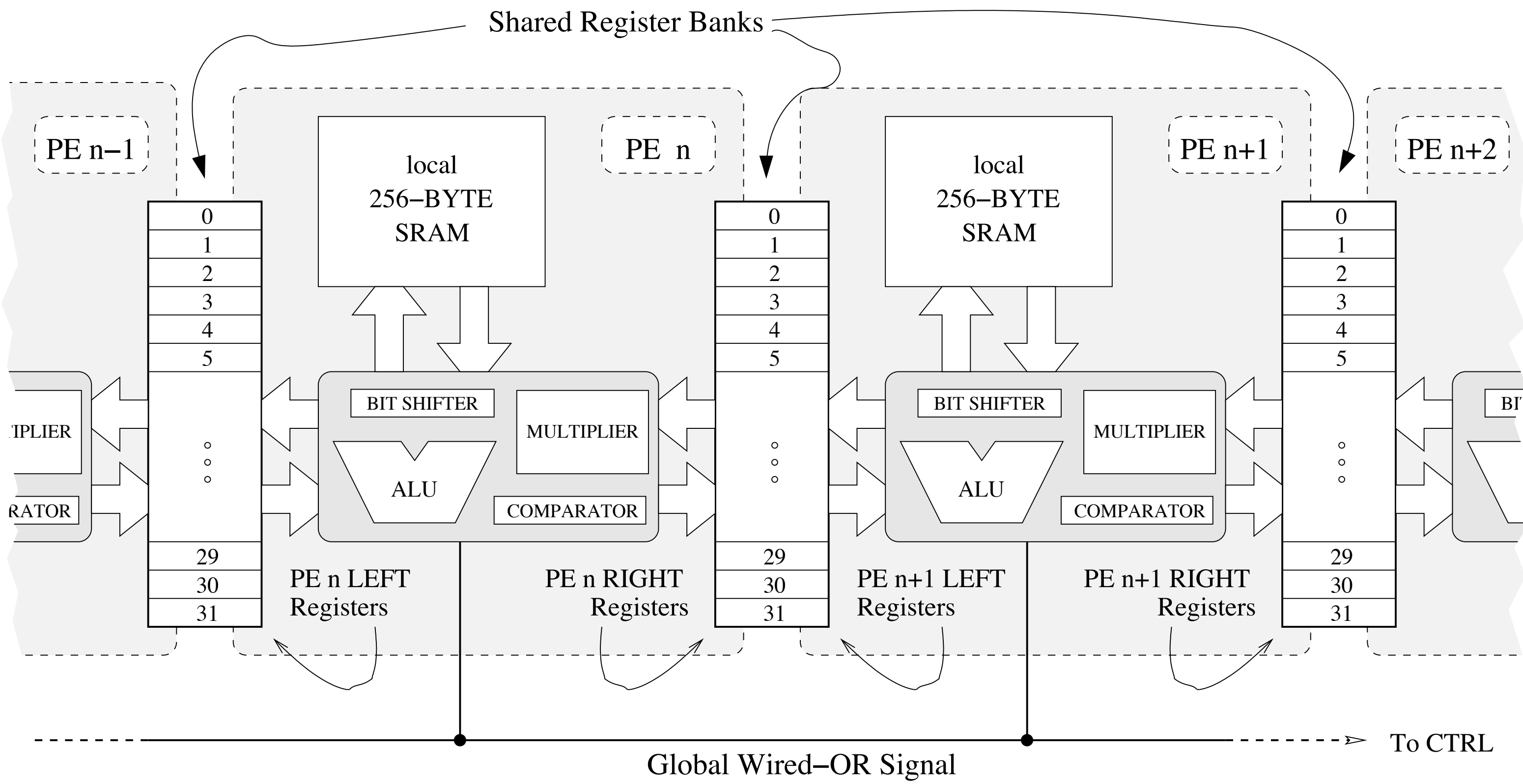
UCSC Kestrel: the board



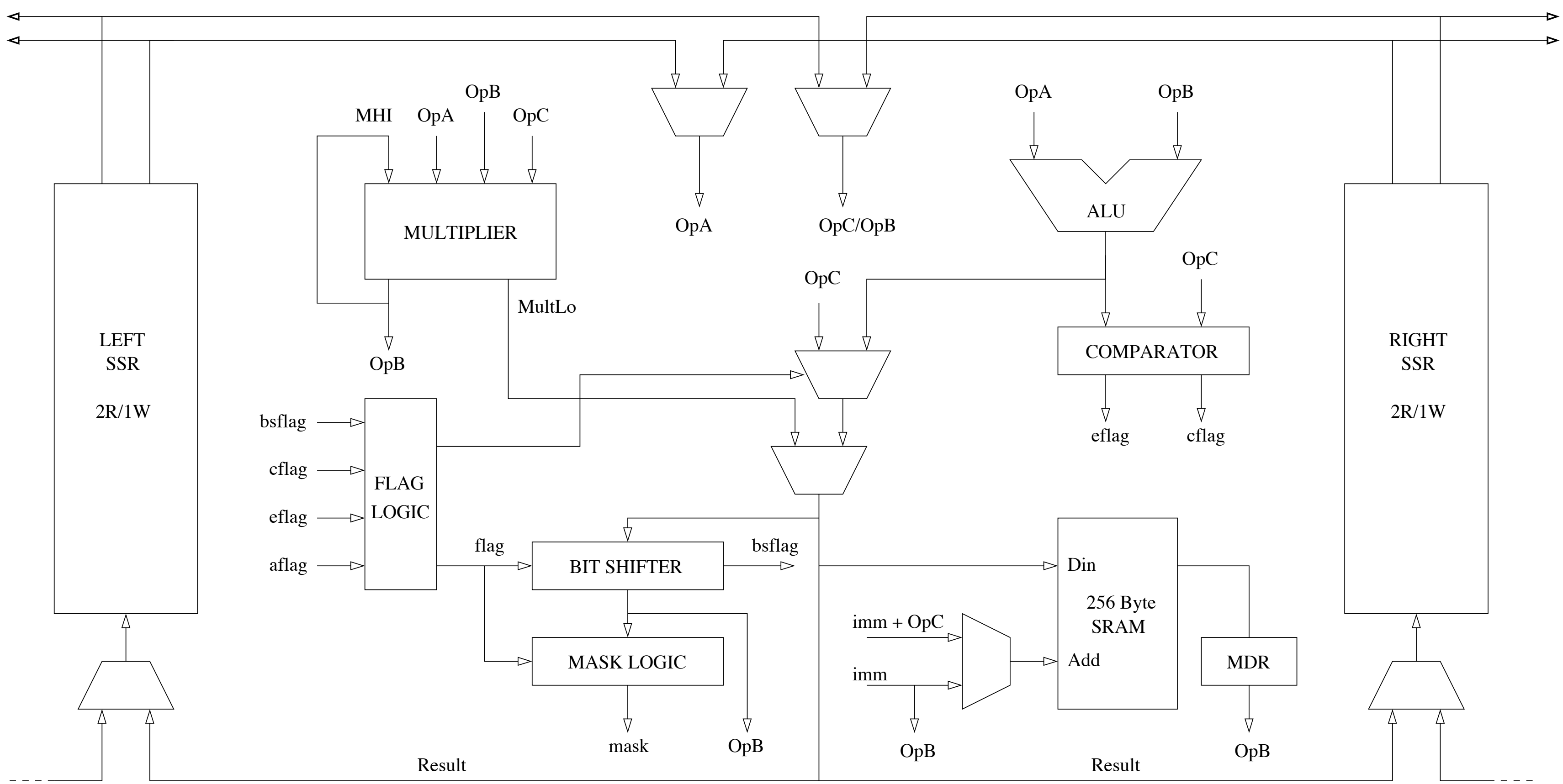
Kestrel's array structure



Kestrel array detail



Kestrel processing element detail



Kestrel ISA

- 8-bit word size, support for multi-precision operations
- Two *systolic shared register* banks per PE, 32 8-bit registers each
- 0-, 1-, 2-, 3-, 4-, or 5-operand instructions - high instruction-level parallelism with multiple functional units
- Load/store architecture
- 256 bytes of local, locally-addressable memory per PE
- Single data addressing mode
- Single-cycle implementation
- No floating-point unit or support
- Official manual: Richard Hughey, *KASM Assembly Manual*, Tech. Rep. UCSC-CRL-98-11, UC Santa Cruz, Sept. 1998 (a copy is provided in our course's website)

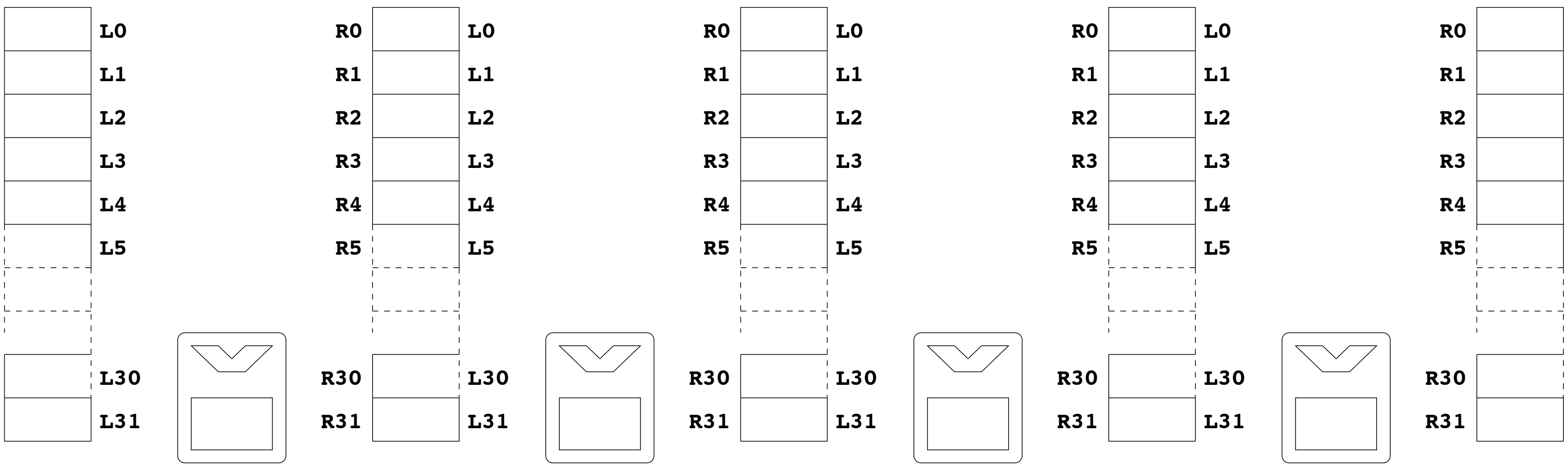
First Kestrel program

```
;***********************************
; Program: kex01.kasm
; First Kestrel program example
; Andrea Di Blas, May 2005
;***********************************
start:    ;  program execution starts here
addzz     L0
add       L0, L0, #3
;***********************************
```

What does this program do?

What does kex01.kasm do?

Assume to run it on a 4-PE array like this:




```
start:  ;  program execution starts here
addzz   L0
add     L0, L0, #3
```

Assembling the program

To install the Kestrel environment:

- Download the tar files **`kestrelClient.tar.gz`**, **`kestrel_win32.tar.gz`**, and **`newkasm.tar.gz`** from our web page.
- Follow the instructions to **`make`** the assembler, **`newkasm`**, and the client/simulator/debugger, **`kestrel`** (look at the README files). Make sure to **`make clean`** before **`make`**.

To assemble the program:

```
$ newkasm kex01.kasm
```

The output is an object (executable) called **`kex01.ko`**

Running the program

Simulating an array with 4 PEs:

```
$ kestrel kex01.ko kin kout 4
```

Where **`kin`** is the input file name and **`kout`** is the output file name. Both need to be specified even when not actually used (use any dummy input file for **`kex01`**).

If you forget the syntax:

```
$ kestrel -h
```

Except that the board option is not available (simulator only)

Addition instructions

ADD writes in **`Rdst`** the sum of two operands

```
ADD     Rdst, OpB, OpA
```

ADDXX writes in **`Rdst`** sum of an operand with itself

```
ADDXX   Rdst,  [OpA or OpB]
```

ADDXZ writes in **`Rdst`** sum of an operand with zero

```
ADDXZ   Rdst,  [OpA or OpB]
```

ADDZZ writes in **`Rdst`** sum of zero with zero

```
ADDZZ   Rdst
```

Operand A, operand B, operand C

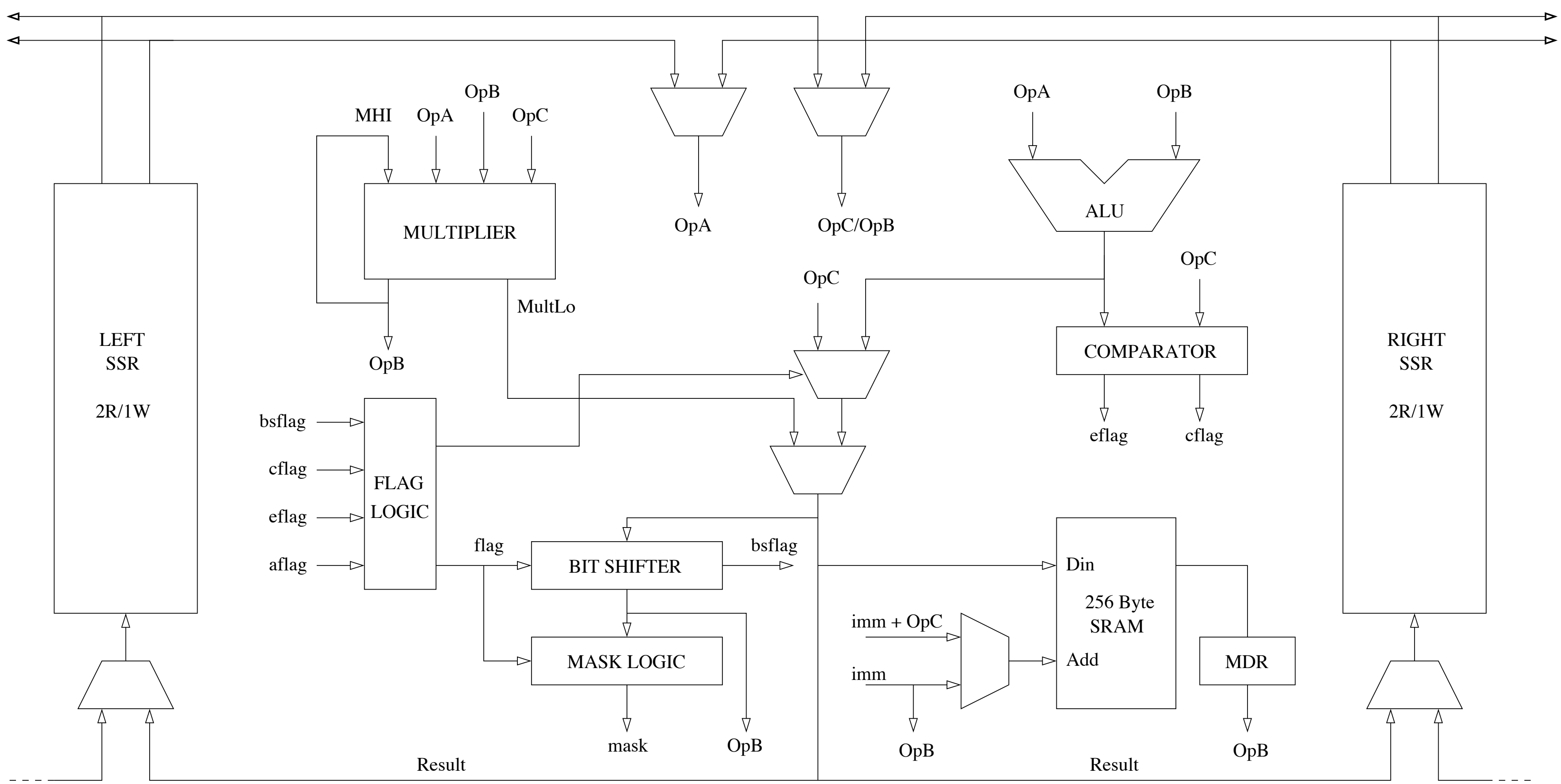
Three operand busses, A, B, and C (fig. on page 2.5).

Operand A and operand C can only be registers.

Operand B can assume one of the following values:

- Operand C (e.g. **`L17`**)
- Sign extension of operand C (e.g. **`sL17`**)
- Memory Data Register (**`mdr`**)
- Sign extension of **`mdr`** (**`smdr`**)
- Multiplier high byte (**`mhi`**)
- Sign extension of **`mhi`** (**`smhi`**)
- Bit shifter (**`bs`**)
- Instruction immediate (e.g. **`#73`**)

Kestrel processing element detail again



Running `kex01` in the debugger

Start the debugger, simulate 4 PEs:

```
$ kestrel -debug kex01.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: end of program detected at instruction 3
kestrel rte: Starting the Kestrel Debugger (kdb)
```

(continued on the next page)

The debugger `main` menu

```
@Main Menu:

 run        - Run your program
 step       - single Step through your program
 examine    - Examine a specific PE
 range      - examine single value across a Range of PEs
 controller - examine state of the controller
 list       - List most recent instructions
 breakpoint - set, change or remove Breakpoints
 format     - change the display format
 precision  - change the Precision
 settings   - print current format information
 history    - print history buffer of commands
 menu       - display this Menu
 quit       - Quit kdb
 dump       - Single step program, dumping state to file 'state.dump'

@Main Menu:
kdb>
```

Stepping through `kex01`: the `range` menu

```
kdb> range
@Main Menu > Range Menu:

 reg <#> [- <#>]  - examine a range of registers in each SSR
 sram <#> [- <#>] - examine a range of sram locations in each PE
 masklatch        - view the maskLatch
 minlatch         - view the minLatch
 bslatch          - view the bsLatch
 mdrlatch         - view the mdrLatch
 clatch           - view the cLatch
 multhilatch      - view the multhilatch
 eqlatch          - view the eqLatch
 menu             - display this Menu
 back             - back to Main menu
 quit             - Quit kdb

kdb>
```

NOTE that **`range`** is reliable, while **`examine`** reports incorrect values - do not use.

Looking at register 0

```
kdb> reg 0
kestrel rte: get register state: end of program detected at instruction 455
kestrel rte: running program get register state at 6
kestrel rte: end of program reached at 455
QIN DATA: 1 QOUT DATA: 256

  reg   0 values from 0 to 3 are:
  0:   0,   0   1:   0,   0   2:   0,   0   3:   0,   0

kdb>
```

Stepping through kex01

```
kdb> b
@Main Menu:
kdb> step
kestrel rte: Running kex01.ko at 0
kestrel rte: Status Register: kestrel rte: program stepped, now at address 0
kdb> step
kestrel rte: Status Register: kestrel rte: program stepped, now at address 1
kdb> step
kestrel rte: Status Register: kestrel rte: program stepped, now at address 2
kdb>
```

Here's the code again:

```
start:                          ;  address 0
addzz        L0                 ;  address 1
add          L0, L0, #3         ;  address 2
[Program Termination]           ;  address 3
```

Stepping through `kex01`

```
kdb> reg 0
QIN DATA: 11 QOUT DATA: 0
kestrel rte: get register state: end of program detected at instruction 455
kestrel rte: running program get register state at 6
kestrel rte: end of program reached at 455
QIN DATA: 1 QOUT DATA: 256


  reg   0 values from 0 to 3 are:
  0:   0,   0   1:   0,   0   2:   0,   0   3:   0,   0


kdb>
```

Stepping through `kex01`: run to end

```
kdb> b
@Main Menu:
kdb> step
kestrel rte: Status Register: kestrel rte: program stepped, now at address 2
kdb> run
kestrel rte: continuing execution of kex01.ko at 2.
kestrel rte: end of program reached at 3
QIN DATA: 3 QOUT DATA: 0
kestrel rte: program kex01.ko completed at 3 with 0 output bytes.
kestrel rte: program used only 0 of 2 input data bytes.
kdb> range
@Main Menu > Range Menu:
```

[The Range Menu]

Looking at final content of register 0

```
kdb> reg 0
kestrel rte: get register state: end of program detected at instruction 455
kestrel rte: running program get register state at 6
kestrel rte: end of program reached at 455
QIN DATA: 1 QOUT DATA: 256

  reg   0 values from 0 to 3 are:
  0:   3,   3   1:   3,   3   2:   3,   3   3:   3,   0

kdb>
```

What does `kex01.ko` look like?

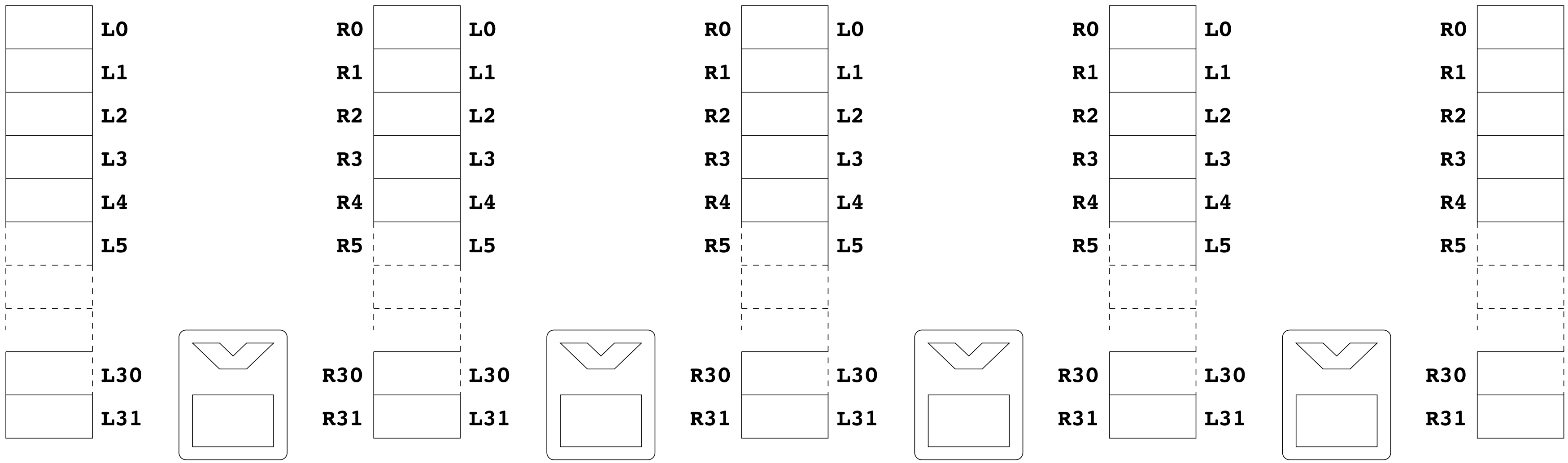
```
$ cat kex01.ko
b01000000000020000218001
bc9000000000020000018000
e09001c00000320000018000
b01000000000020000118003
$
```

Second Kestrel program

```
;*********************************** 
; Program: kex02.kasm
; Kestrel program example
;***********************************
start:   ; program execution starts here
addxz  L0, #1
add    R0, L0, #1
add    R0, L0, #1
add    R0, L0, #1
;***********************************
```

What does this program do?

Tracing kex02




```
addxz   L0, #1
add     R0, L0, #1
add     R0, L0, #1
add     R0, L0, #1
```

Loops

The controller has a loop counter stack that supports up to 15 nested loops.

BEGINLOOP pushes a new loop counter onto the stack

```
beginloop  CImm
```

Where **`CImm`** is a 16-bit immediate value, unsigned.

ENDLOOP decrements the current loop counter (TOS) and branches back to the instruction following the corresponding **`beginloop`** if the count is greater than zero. If the count is zero, pops it off the stack and proceeds with the instruction following the **`endloop`**.

```
endloop
```

NOTE: the loop counter is NOT available as a variable. Also, the branch target address for **`endloop`** is encoded into the controller's 16-bit immediate field as an *absolute* reference to the instruction number, so no loop can *begin* at addresses higher than 64K.

Example of loop

```
;***********************************
; Program: kex03.kasm
; Kestrel program example
; Same as kex02 but with a loop
; Andrea Di Blas, May 2005
;***********************************
start:   ; program execution starts here
addxz  L0, #1
beginloop 3
    add  R0, L0, #1
endloop
;***********************************
```

Controller and array immediate

Two different immediate fields:

Controller immediate:

- 16 bit
- unsigned
- syntax: **`12345`** (only used with **`beginloop`**)

Array immediate

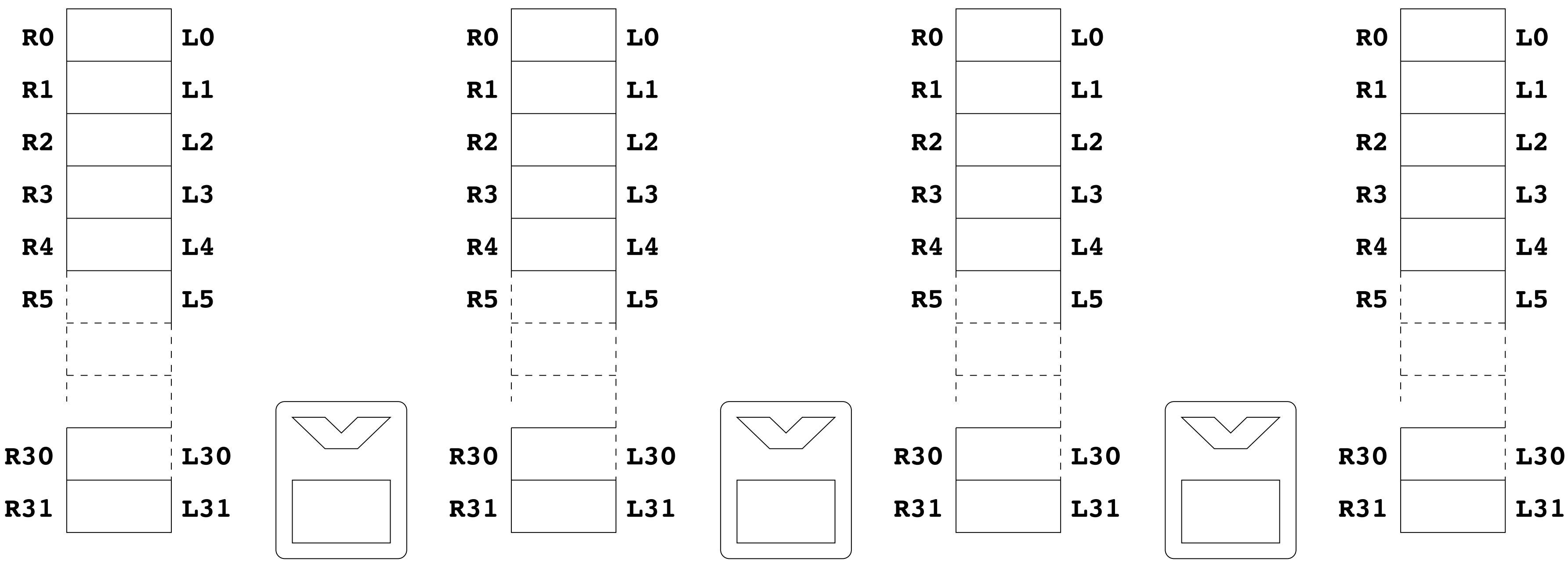
- 8 bits
- unsigned or signed (two's complement)
- syntax: **`#123`**

The MOVE instruction

MOVE move an operand to a destination:

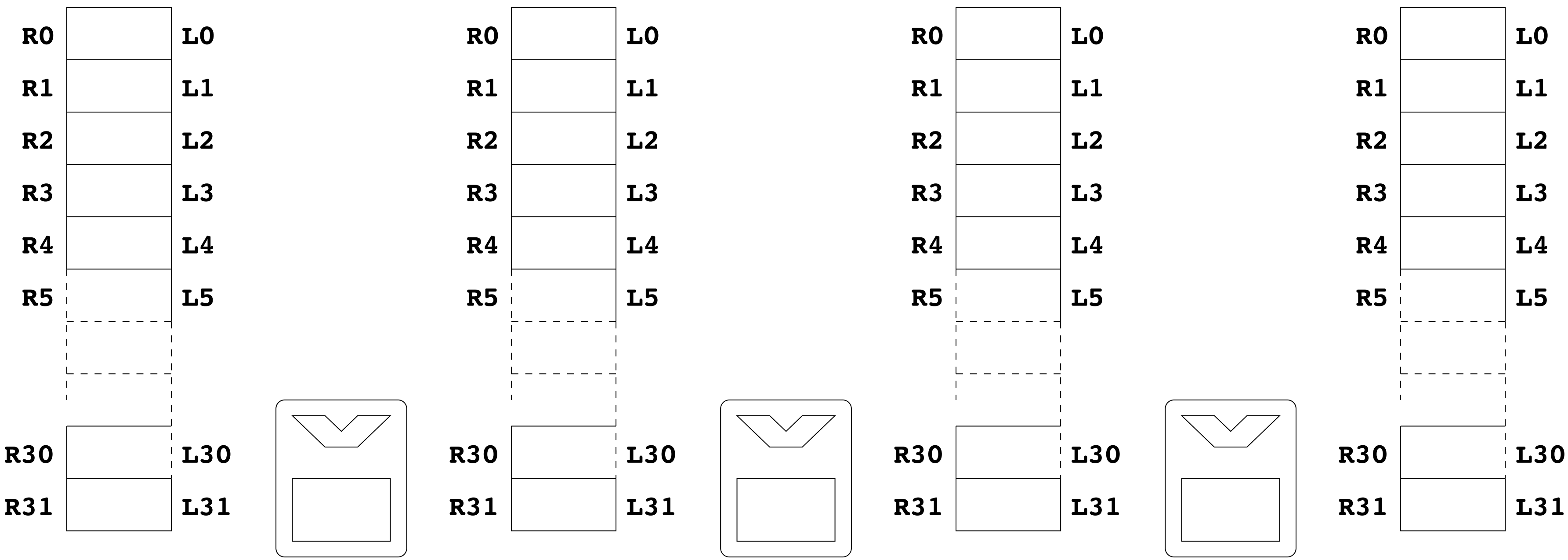
```
MOVE  Dst, OpA or OpB
```

Example: Source and destination are registers on same side:



The `MOVE` instruction: another example

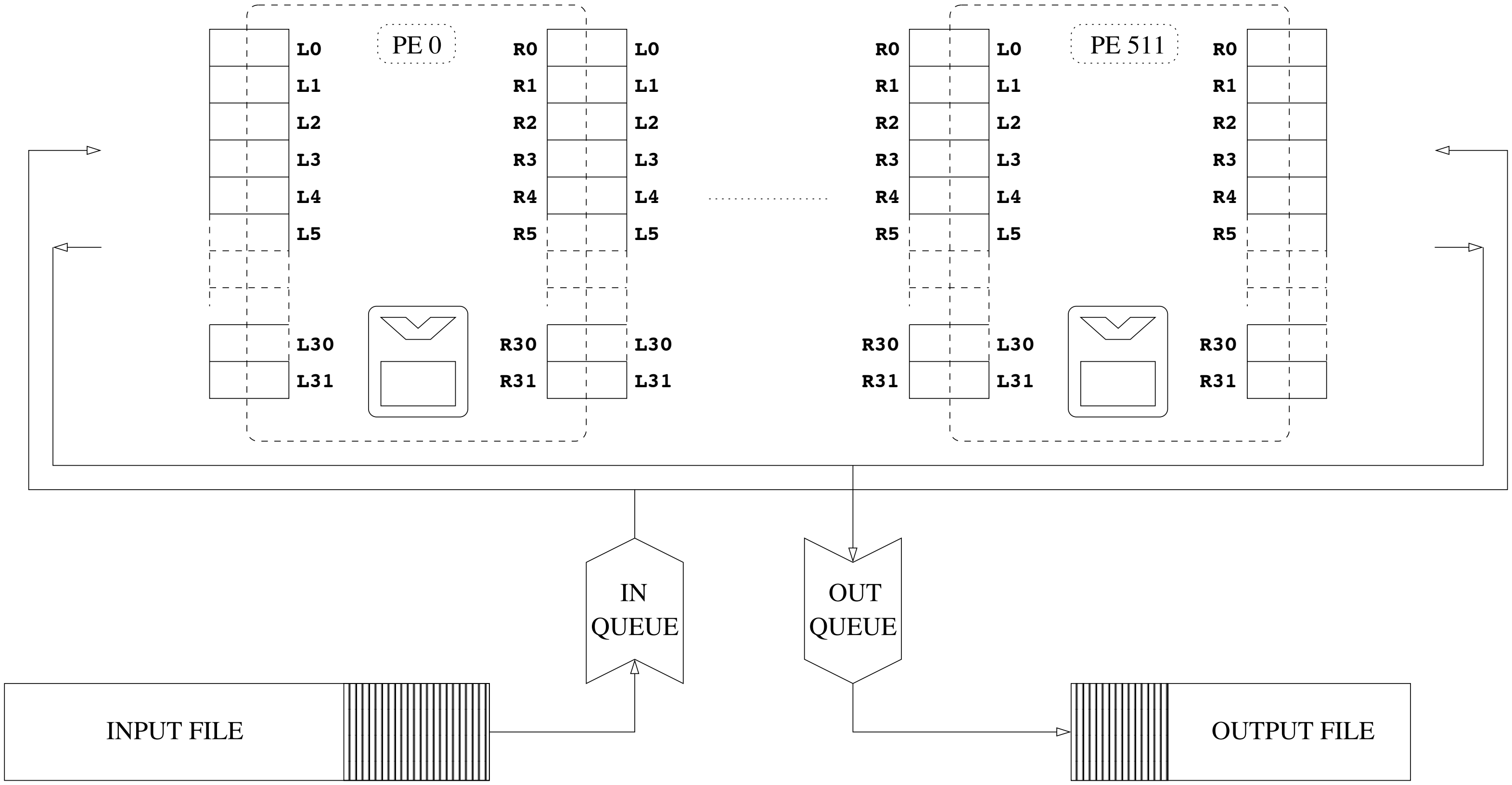
Source and destination are registers on different sides:



What happens at the edges?

Kestrel I/O system

There are always two files: an input file and an output file.



The input file is a FIFO called Input Queue, the output file is a FIFO called Output Queue

File input: `qtoarr`

QTOARR “**Q**ueue **TO ARR**ay” writes the next byte in the input file into the destination register

```
<some instr> Rdst, ...  qtoarr
```

- If **`Rdst`** is a *left* register, the value will be written into the right register bank of the rightmost PE. If **`Rdst`** is a *right* register, the value will be written into the left register bank of the leftmost PE.
- The registers are written regardless of the ***mask*** of the PE being written (explained below).
- Input and output can overlap, so **`qtoarr`** and **`arrtoq`** can appear in the same instruction.

File output: `arrtoq`

ARRTOQ "**ARR**ay **TO Q**ueue" writes the result of the instruction to the output file as well as into the destination register and into the controller's *scratch* register.

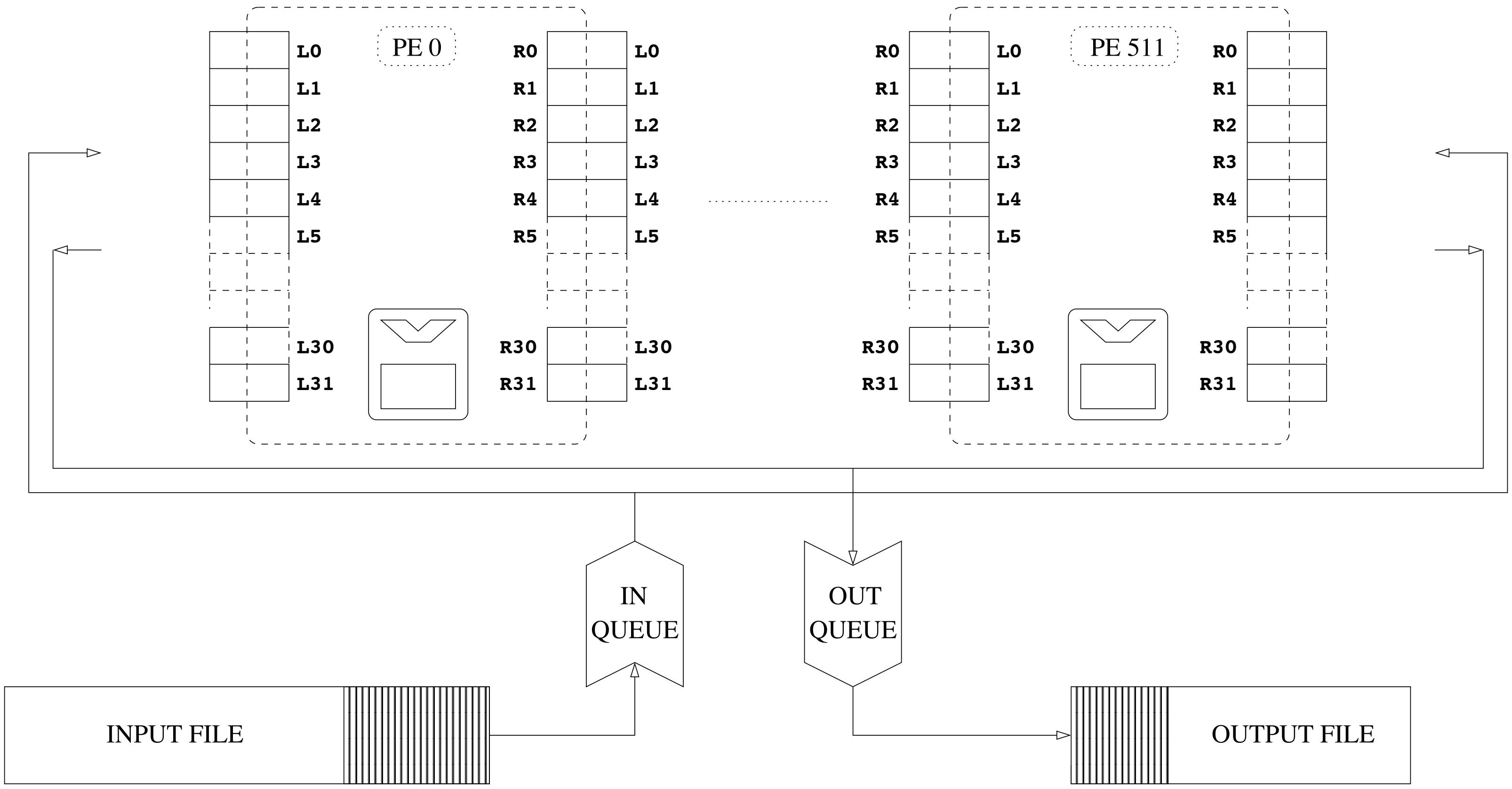
```
<some instr> Rdst, ...  arrtoq
```

- If **`Rdst`** is a *left* register, the value comes from the leftmost PE in the array. If **`Rdst`** is a *right* register, the value comes from the rightmost PE in the array.
- If the outputting PE is ***masked***, no output is produced (explained later).

File input and output overlap

- Input and output can overlap, so **`qtoarr`** and **`arrtoq`** can appear in the same instruction.
- That is the maximum extent of I/O parallelism that we have here, no multiply opened files with parallel axcess

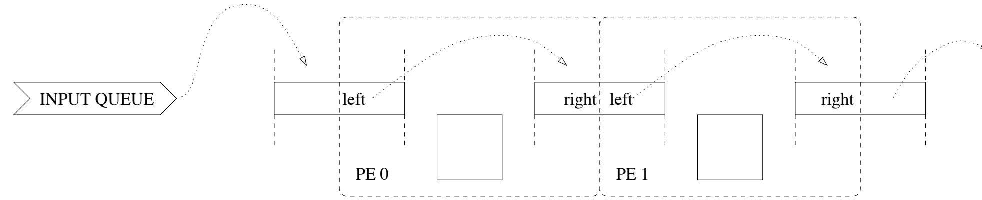
I/O example



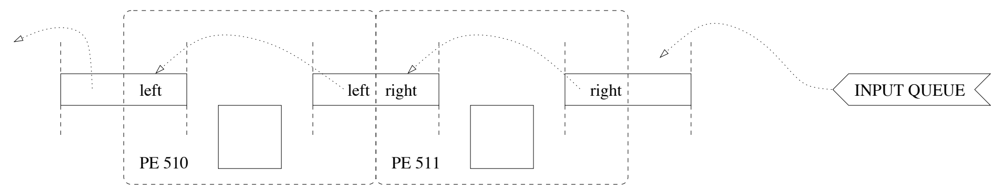

```
1.  ADD   R2, L0, R4, arrtoq
2.  SUB   R2, L0, R4, qtoarr
3.  MOVE  L5, R5, qtoarr, arrtoq
```

Stream I/O details

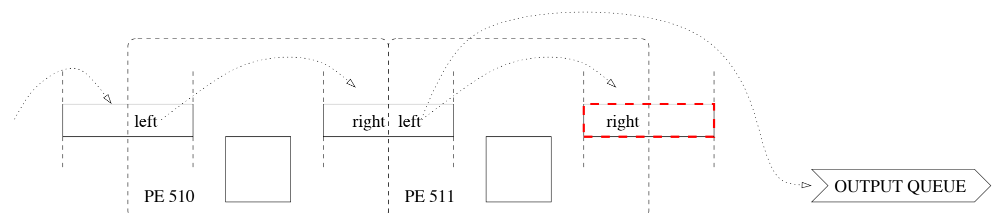
MOVE R0, L0, qtoarr



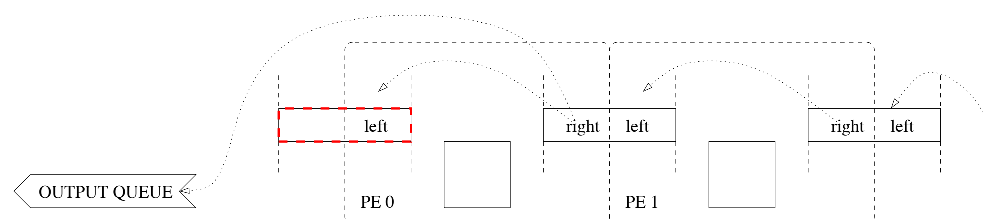
MOVE L0, R0, qtoarr



MOVE R0, L0, arrtoq



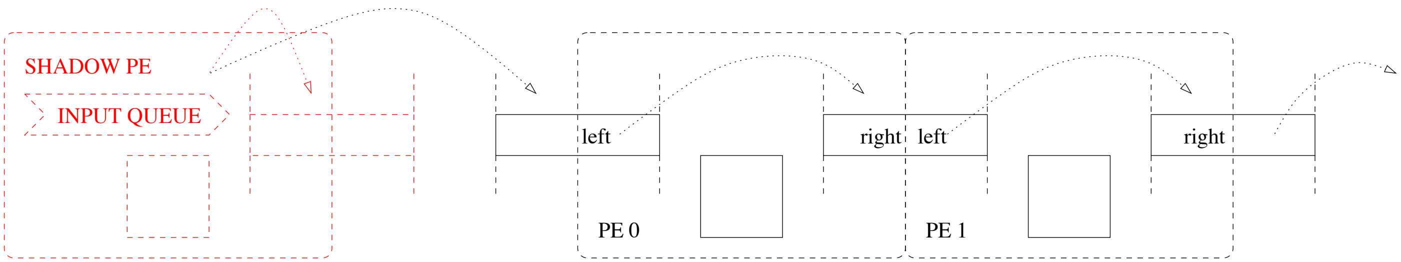
MOVE L0, R0, arrtoq



Stream I/O: output "anomaly"

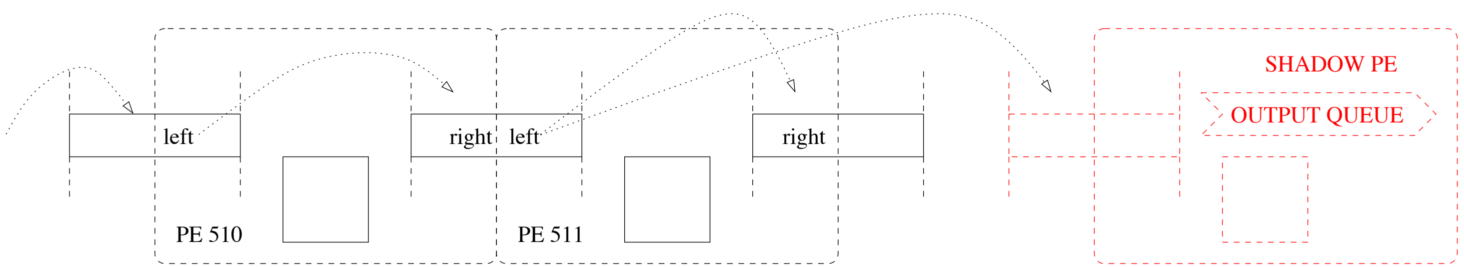
Input can be modeled as if there was an additional "shadow PE" at the end of the array where the data is input from.

```
MOVE  RO, LO, qtoarr
```



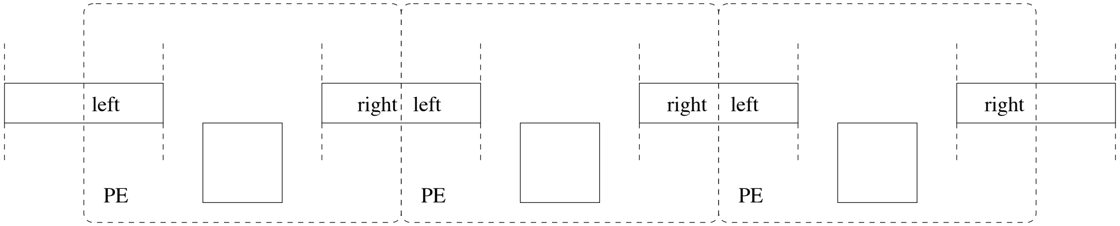
Output, however, behaves differently and counter-intuitively: in this example, the rightmost RO is not written to to output file. The reason is in how inter-chip communication is implemented at the hardware level.

```
MOVE  RO, LO, arrtoq
```

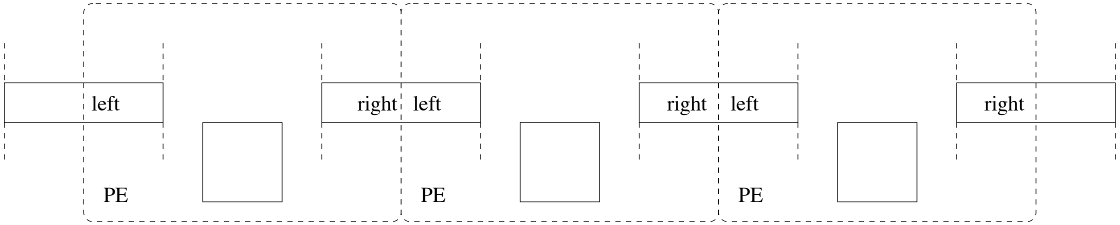


Stream I/O: more examples

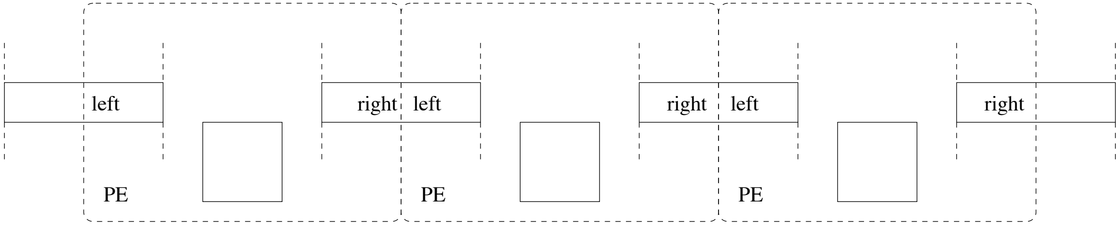
`MOVE  L0, L0, qtoarr`



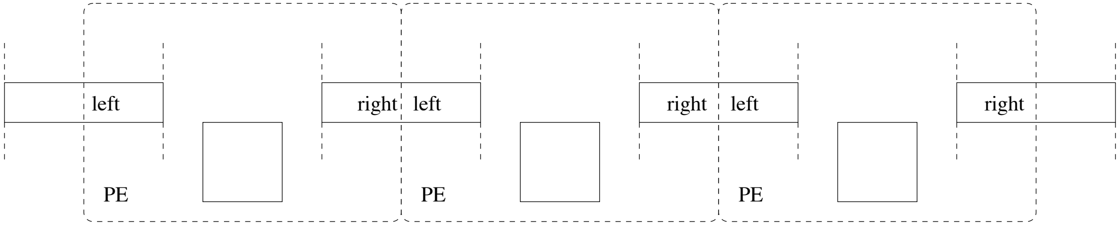
`MOVE  R0, R0, qtoarr`



`MOVE  L0, L0, arrtoq`



`MOVE  R0, R0, arrtoq`



Program with I/O

```
;***********************************
; Program: kex04.kasm
; Kestrel program example
;***********************************
start:
bsclearm
addzz       L0
beginloop   8
    move    R0, L0, qtoarr
endloop
add         R0, R0, #1
beginloop   8
    move    L0, R0, arrtoq
endloop
end:
;***********************************
```

What does this program do?

Input and output file format

The input and output file format can be specified individually for each file with the option:

`-iformat <format>` for the input file, or

`-oformat <format>` for the output file.

where **`<format>`** can be:

- **`decimal`** for ASCII decimal data, newline separated (default)
- **`octal`** for ASCII octal data, newline separated
- **`hex`** for ASCII hex data, newline separated
- **`binary`** for raw binary data

Kestrel I/O file format example

Decimal	Octal	Hex	Raw
33\n	41\n	21\n	!
34\n	42\n	22\n	‘ ‘
45\n	55\n	2D\n	-
56\n	70\n	38\n	8
9\n	11\n	9\n	\t
78\n	116\n	4E\n	N
1\n	1\n	1\n	[1]
2\n	2\n	2\n	[2]
...	...	...	...

Programming assignment 1: PE count

Since when running a program with the simulator we can specify the number of PEs on the command line, how can a *program* figure out *at run time* how many PEs are available?

Write a program that stores in the register L0 of all PEs the number of PEs that are being simulated, assuming it is at most 255. You can only use instructions that we have seen so far in this chapter.

Make sure it works with all corner cases, such as 1 and 255.

PE count example

```
examples$ kestrel -debug kPA01_PEcount.ko kin kout 181
...
@Main Menu:kdb> run
...
kdb> range
...
kdb> reg 0
...
 reg   0 values from 0 to 180 are:
0: 181, 181   1: 181, 181   2: 181, 181   3: 181, 181
...
```

Programming assignment 1: PE alternate

Write a program that reads 8 numbers from a file, one byte each, and then writes them to the output file alternatively first the first one read, then the last one read, then the second one read, then the second to last one read, and so on. Example:

Input file: 1 2 3 4 5 6 7 8

Output file: 1 8 2 7 3 6 4 5

You can choose the number of PEs that should be used, but you should use as many PE as possible to make your code as simple and efficient as possible (specify NPES in the README file included with your submission).

Code grading criteria

25% Code compiles and runs, and runs simple input files correctly.

25% Code runs corner cases correctly.

25% Code runs “secret” input files correctly.

25% General code *quality* and *clarity*.

Clean code

- Is **easy to read**, to understand, to debug.
- Has **meaningful variable names** and appropriate **comments**: explain the "why", summarize the "what."
- Is **modular**, but without being too modular. e.g. some meaningful macros are good, too many macros with cryptic names are bad.
- Is **well aligned**, has columns that don't exceed 80 characters (Oracle, Amazon, and Google C Coding Standard!)
- **Doesn't jump around** too much and has clear entry and exit points from functions/blocks.

Code efficiency

- **Main principle: code should use all available parallelism.**
- **HOWEVER**, in assembly in general and especially in Kestrel, often efficiency and clarity go against each other. If you use all possible optimizations offered by ILP, your code becomes easily unreadable. Therefore, optimizing all the way is the right approach only when it's worth it, usually in inner loops.
- For example, merging the PEID assignment with input file read does save instructions, but when you have to deal with much bigger pieces of code, saving 512 clock cycles may be less important than keeping these two basic operations clearly separate.