EE382A – Spring 2025

Chapter 9:
Kestrel's Instruction Set Architecture (5 of 5)

Kestrel's ISA, fifth part

- Parallel communication operations
- Parallel reduction operations
- The wired-OR global reduction network
- Control transfer instructions: jump and jumpwor
- The bit-serial log-reduction mesh
- Additional instructions
- Binary instruction format

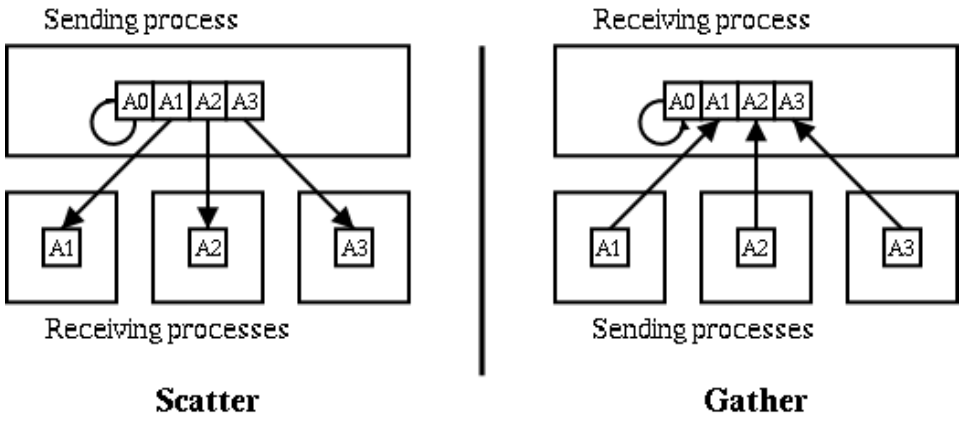
Parallel communication operations

- Point to point

- Global

Parallel communication operations

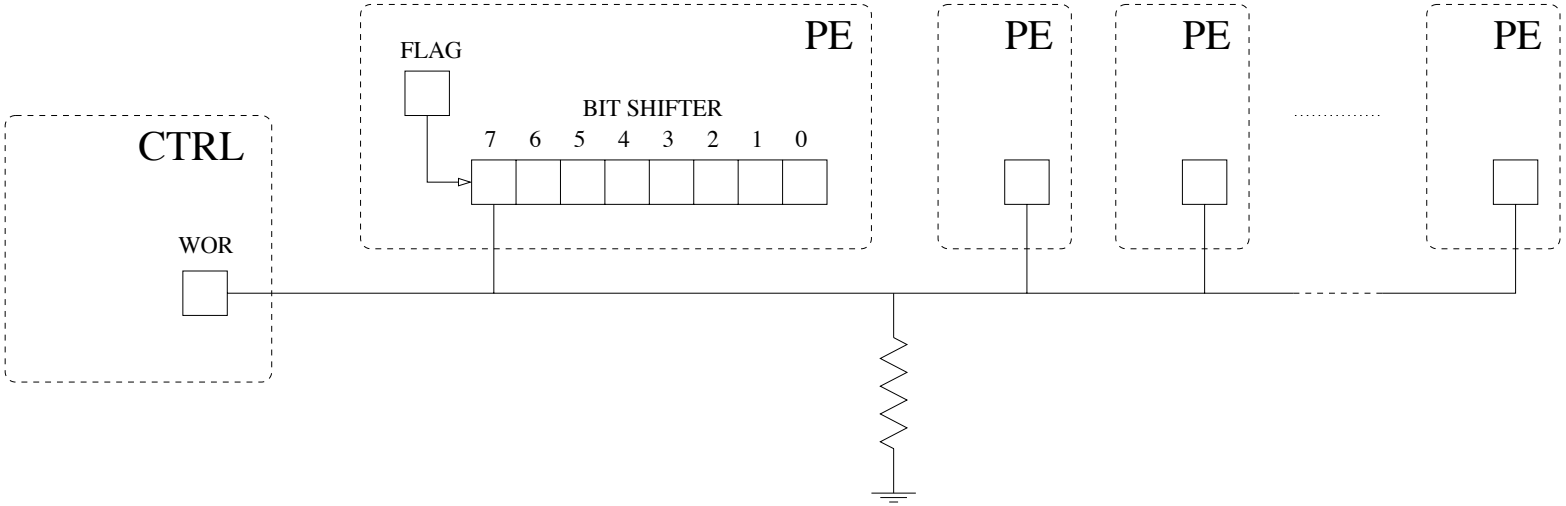
- Parallel gather/scatter



- "Normal" gather/scatter

Parallel reduction operations

The wired-OR



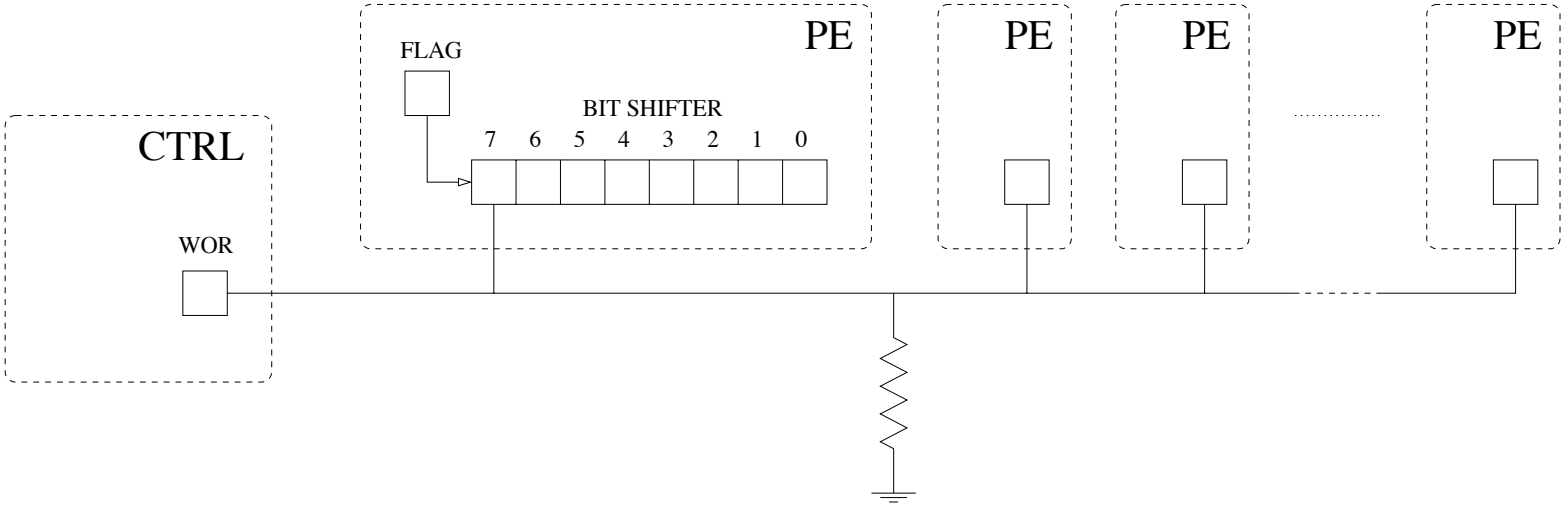
The most significant bit of the bit shifter in all PEs is connected to a wired-OR line. Global OR-reduction: if at least one PE raises the wired-OR, the controller reads a 1.

Special instructions change the BS but not the mask.

These instructions execute conditionally (only if the PE is active).

Due to pipelining, there must be 2 instructions between setting the wired-OR and reading it.

The wired-OR



The wired-or is gated by the mask. That's why a 1 in the BS due to a condition (which turns off the mask) does not affect the wired-or.

Wired-OR instructions

BSCONDRIGHT right-shifts the bit shifter, shifting the flag into the msb. Use to set the WOR. Occurs only in unmasked PEs. Does not change the mask.

```
<any flag-setting instr.>  bscondright
```

BSCONDLEFT left-shifts the bit shifter, shifting the flag into the lsb. Use to restore the bit shifter. Occurs only in unmasked PEs. Does not change the mask.

```
<any flag-setting instr.>  bscondleft
```

Control transfer: `jump` and `jumpwor`

JUMP unconditionally jumps to the specified label. It is a controller instruction.

```
jump Label
```

JUMPWOR jumps to the specified label if the wired-OR signal coming from the array is 1.

```
jumpwor Label
```

`Label` is an absolute instruction address, and it is encoded in the controller's 16-bit immediate field. Therefore no jump can target an instruction at an address higher than 64K.

How to set the wired-OR?

Example of jump on wired-OR

```
addzz    L20       ltc L19
addxz    L20, #3   ltc L18 cmp fbinv bspush       ; if(r < 4)
  addxz    L1,  L1  equalc L1 fbinv bscondright ; raise wor
  nop
  nop
  fbaco, bscondleft, jumpwor JUST_LOAD ; skip row convol.
bspop
 ...  ; row convolution
JUST_LOAD:
bspop
```

NOTE: when jumping from within conditional statements, must make sure to re-align the bit shifter (i.e. **bspop**) enough times.

Mistake to avoid

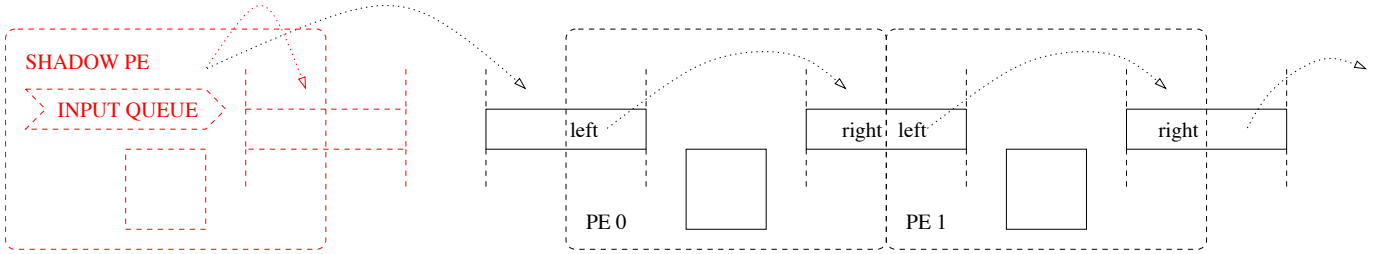
DO NOT MIX the active set with setting the wor!

```
; Intention: IF (all L4 == L16) then jump to RESEC

addxz L4, L4 equalc L16 fbinv bscondright
  nop
  nop
fbaco bscondleft jumpwor RESEC
jump FINISHED
RESEC:
FINISHED:

; Is really doing
IF (at least one L4 == L16) then jumpwor RESEC
```

Conditional execution and input file read

Conditional execution only applies to the PEs, not to the controller. Therefore the input file read instruction **`qtoarr`** *will execute even if the corresponding end PE is turned off!* This is easily explained by the fact that the ``shadow register'' doesn't have a mask and cannot be turned off. Example:

```
addxz L10, L10 equalc L10 fbinv bspush   ; turn off all PEs
  move  R0, L0, qtoarr
bspop
```



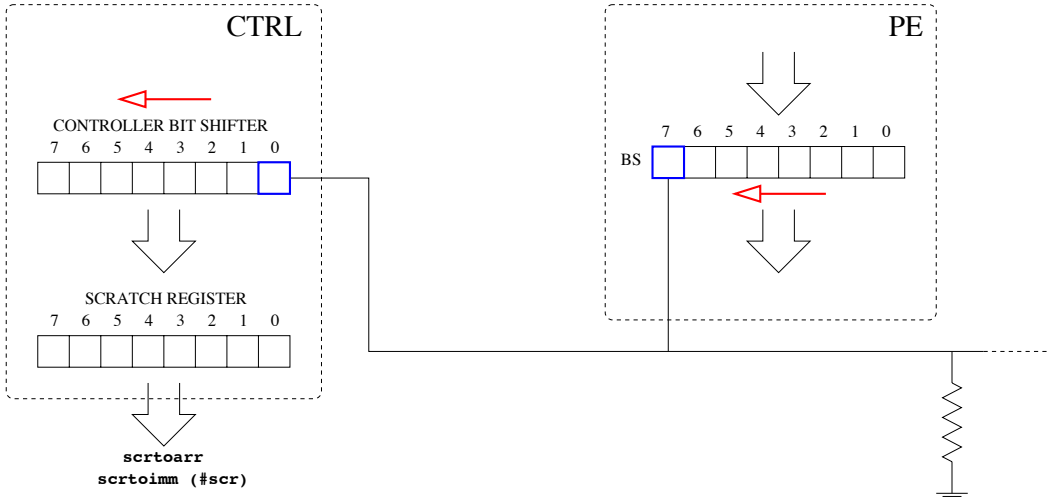
Will write the next byte from the input file into register L0 of the leftmost PE, despite the fact that all PEs are turned off.

Conditional input file read

The following structure conditionally skips the **`qtoarr`** instruction:

```
addxz   L7, L7 ltc L9
addxz   L6, L6 ltc L8 cmp fbinv, bspush        ; IF (NB >= TOTB)
  addxz L0, L0, equalc L0, fbinv, bscondright ;   raise wor
  nop                                          ;
  nop                                          ;   and skip input file read
  fbaco, bscondleft, jumpwor  C0P0A            ;
bsnot                                          ; ELSE
  [...]
  move    R0, L0, qtoarr                       ;   read initial col value
  [...]
bspop
[...]
C0P0A:
bspop                                          ; re-align the bit-shifters
```

The controller's bit shifter and the wor



Turn on one PE only

Save current BS: **`bs`** can be used as an operand (OpB)

Load BS with byte to send without affecting the mask (**`bscondlatch`**)

Start a loop for 8 times (for a full byte) that does:

- Shift left WOR bit into controller's bit shifter lsb (**`cbssleft`**)
- Shift left the PE's bit shifter, without affecting the mask (**`bscondleft`**)

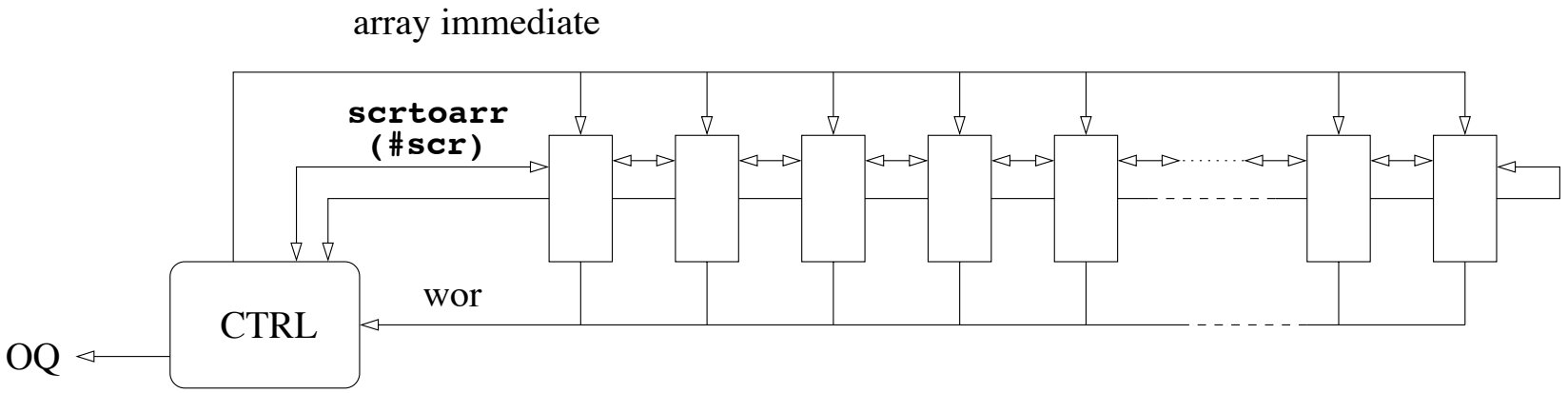
To use the byte received, copy it into the scratch register (**`cbstoscr`**)

Restore the BS (**`bscondlatch`**)

The controller's bit shifter in the debugger

- In the debugger menu controller->state, the value of CBS (WOR) is NOT the value of the wired-or signal, but the value of the actual register, which is 0 even when wor=1, unless one does a **`cbssleft`**.

Wired-OR as point-to-point or broadcast communication



Bit-serial channel:

- Bandwidth is 3 clock cycles per bit.
- Latency is independent of PE position.

Example of wired-OR used to send a byte from a PE to the controller

```
move  L2, bs                      ; save the BS in L2
read(L19)
move  L13, mdr, bscondlatch ; BS <- CPOS (to send)
BEGINLOOP 8
  nop
  cbssleft                         ; CBS <- BS
  fbaco bscondleft
ENDLOOP
cbstoscr                          ; SCR <- CBS
addxz  L2, L2, bscondlatch  ; restore the BS
```

Two bit shifter's latch instructions

There are two instructions that store a value into the bit shifter:

BSLATCH stores into the BS the value of the result (NOTE that move is an ALU instruction):

```
<ALU/MULT/COMP instr.>  bslatch
```

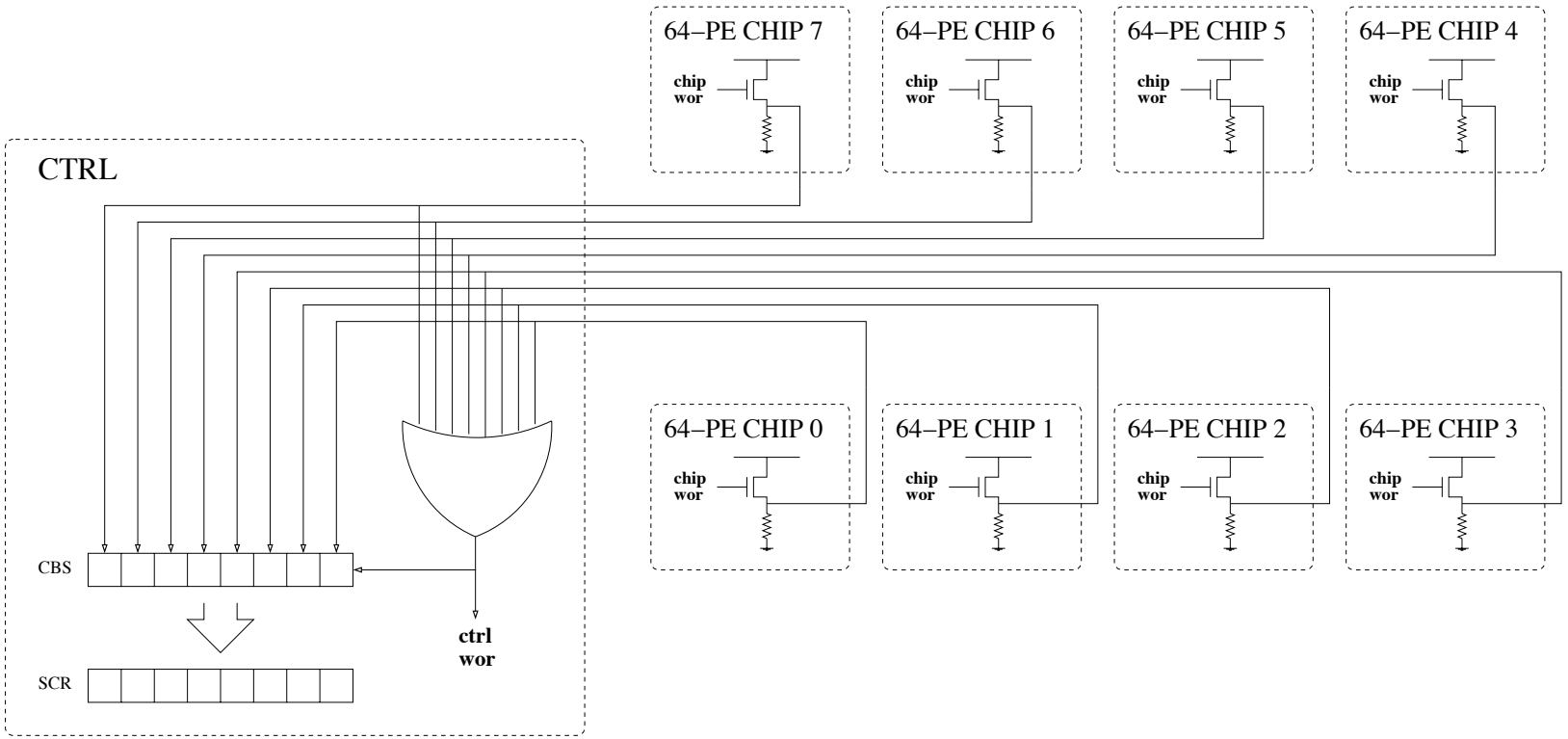
- Occurs in all PEs regardless of the mask.
- The mask is evaluated.

BSCONDLATCH stores into the BS the value of the result

```
<ALU/MULT/COMP instr.>  bscondlatch
```

- Occurs only in active PEs.
- The mask is not affected.

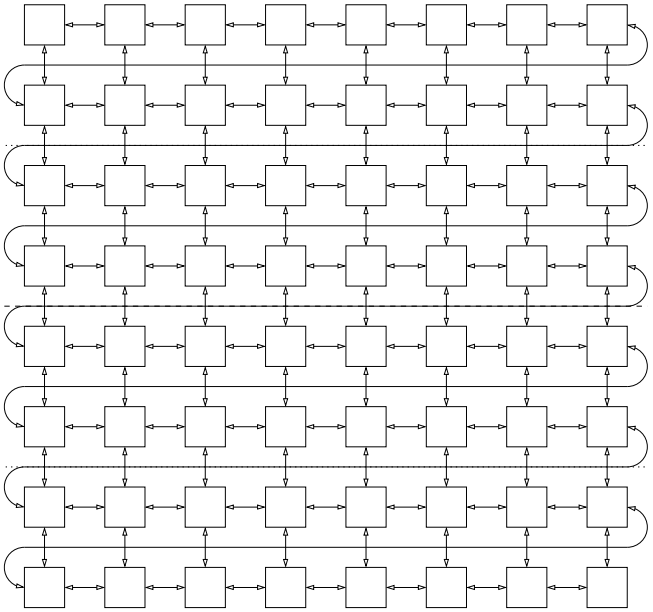
The actual wired-OR connection



CBSLOAD loads the controller's bit shifter with the 8 wired-OR signals

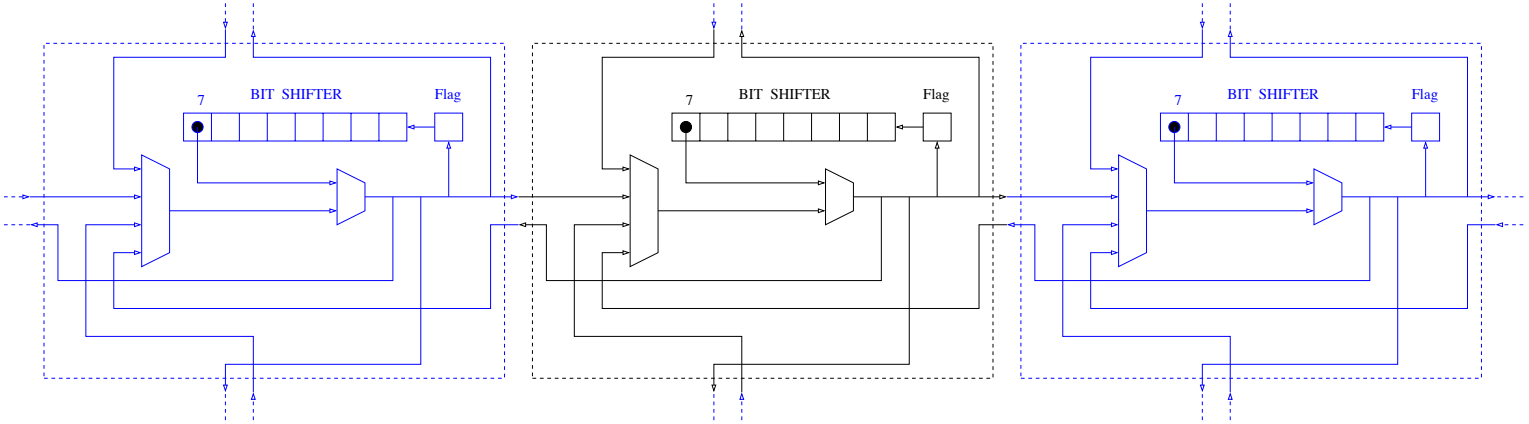
```
cbsload
```

The bit-serial mesh connection



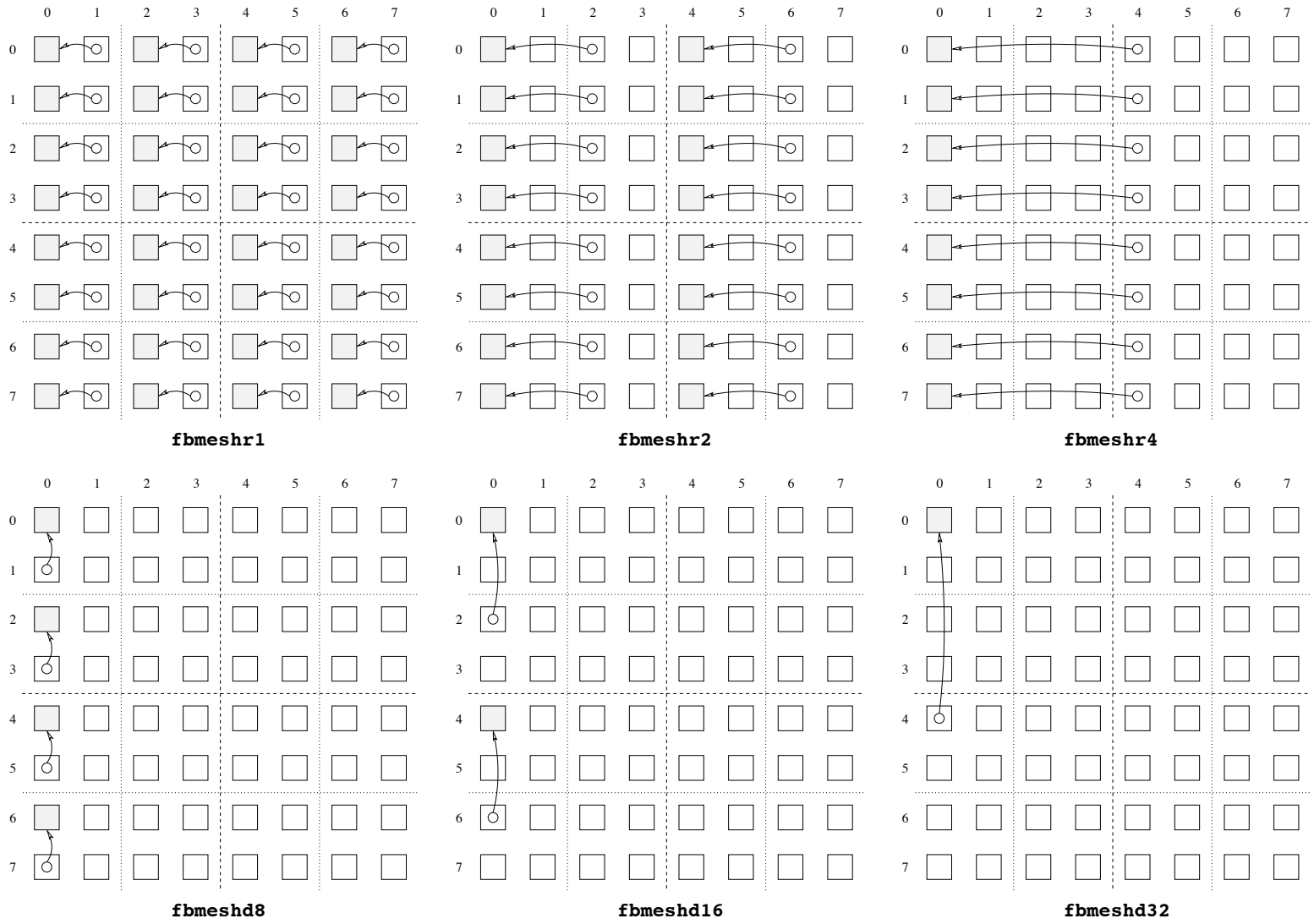
- Each 64-bit PE chip is organized into eight rows of eight PEs, with each PE connecting to the neighboring PEs to the left and to the right on the same row, and with the two PEs above and below.
- PEs at the beginning or end of a row connects to the last PE of the previous row or the first PE of the next row.

The bit-serial mesh - PE detail



- The mesh allows communications between the bit shifter of PEs powers of two apart on the same chip.
- Each PE can either transmit the most significant bit of its bit shifter, or the bit received from a neighboring processor (in the same clock cycle).
- The choice depends on the PE's position in the mesh.

Log-reduction using the mesh

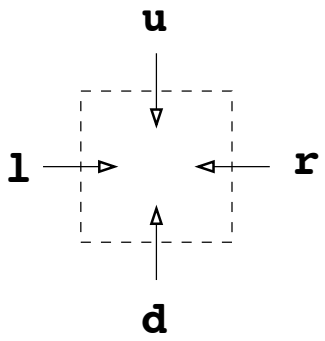


Summary of mesh instructions

All mesh instructions have the following syntax

```
fbmesh sDir dist
```

where **`sDir`** is the direction of the source



and **`dist`** is the distance in number of PEs as if they were all in a line, with respect to the direction specified. Valid values are 1, 2, or 4 if **`sDir`** is left or right, and 8, 16, or 32 if **`sDir`** is up or down.

Communication in opposite directions is symmetric.

Mesh example: count PEs with 1 in LREG (1/2)

```
addzz   L0                          ; counter <- 0
move    L12, bs                     ; save BS
move    L9, #1                      ; used for ANDing
;------------------  MESH1  ------------------------- (1 bit
   read)
addxz   $LREG, $LREG, bscondlatch ; BS <- L10
fbbs0 bscondright       ; (FLAG<-BS[0],BS[7]<-FLAG)=(BS[7]<-
   L10[0])
fbmeshr1 bscondleft                 ; BS[0] <- meshr1
and     L1, bs,  L9                 ; L1 <- 0000,000m
and     L2, $LREG, L9               ; L2 <- 0000,000b
add     L0, L1,  L2                 ; counter <- m + b
```

Mesh example: count PEs with 1 in LREG (2/2)

```
;------------------  MESH2  ------------------------ (2 bit
   read)
addxz   L0, L0, bscondlatch        ; BS <- counter
fbbs0 bscondright        ; (FLAG<-BS[0],BS[7]<-FLAG)=(BS[7]<-L0[0])
fbmeshr2 bscondleft                ; BS[0] <- meshr2
and     L1, bs, L9                 ; L1 <- 0  0  0  0,  0  0  0  m0
fbbs0 bscondright                  ;
fbbs0 bscondright                  ; BS[7] <- L0[1]
fbmeshr2 bscondleft                ; BS[0] <- meshr2
and     L2, bs, L9                 ; L2 <- 0  0  0  0,  0  0  0  m1
mult    L2, #2, L2                 ; L2 <- 0  0  0  0,  0  0  m1 0
or      L1, L1, L2                 ; L1 <- 0  0  0  0,  0  0  m1 m0
add     L0, L1, L0                   ; counter <- counter + meshr2
;------------------  MESH4  ------------------------ (3 bit
   read)
addxz   L0, L0, bscondlatch          ; BS <- counter
fbbs0 bscondright                    ; (FLAG <- BS[0], BS[7] <-
   FLAG) = (BS[7] <- L0[0])
          ...
```

Notes on the mesh example

- The bit shifter's state is preserved in **`L12`**.
- **`bscondlatch`** must be used so that **`mask`** is not changed based on the data value.
- Which makes saving the BS pointless since all PEs must be turned on anyway, right? Or should we **`FORCE`** something?

New instructions:

FBBS0 Flag bus gets bit shifter bit zero

`fbbs0`

FBBS7 Flag bus gets bit shifter bit seven (not in the example)

`fbbs7`

Additional instructions to set the flag bus

`fbaco` flag bus set to ALU's carry out.

`fbats` flag bus set to ALU's true sign.

`fbbsnor` flag bus set to the NOR of the bits in the bit shifter.

`fbcbo` flag bus set to the comparator's borrow out.

`fbclatch` flag bus set to ALU's carry latch.

`fbcmsb` flag bus set to the comparator's msb.

`fbcts` flag bus set to the comparator's true sign.

`fbeq` flag bus set to the comparator's equal output.

Note that **`fbeqlatch`** is the latch that in a multi-precision operation says that all bytes have been the same so far, **`fbeq`** says that the *current* bytes in the comparator are equal.

`fbminlatch` flag bus set to the comparator's *min* latch.

`fbwor` flag bus set to the wired-OR (on same chip only?)

The MOVEC instruction

This instruction is used to leverage instruction-level parallelism, used to move stuff when the ALU is used to produce a flag at the same time.

MOVEC Move register operand C to destination

Example:

```
movec sub L2, L3, mdr, R2, fbats, bspush
```

subtracts **mdr** from **L3**, places the ALU's true sign on the flag bus and pushes it onto the bit shifter, turning off PEs for which **L3** is less than **mdr**, and at the same time copies **R2** into **L2**.

Note that SRAM base plus register addressing mode is not available during a **movec**.

Selection: unsigned mod 256

MODMAXC Maximize with C mod 256: **`Rdst`** gets the maximum of the mod-256 unsigned result and an unsigned operand (**`OpC`**).

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

MODMINC Minimize with C mod 256: **`Rdst`** gets the minimum of the mod-256 unsigned ALU result and an unsigned operand (**`OpC`**).

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

Additional bit shifter instructions

All the following instructions occur in all PEs regardless of the mask.

BSAND ANDs the flag and the msb of the bit shifter. Note that both bits are asserted LOW. The mask is changed.

BSCLEAR clears the bit shifter, but does not change the mask.

BSFLAGMASK sets the mask to the flag bus value.

BSNOTMASK complements the mask.

BSSET sets the msb of the bit shifter to the flag bus value. Changes the mask.

One last controller instruction

JDCNTNZ ``**J**ump and **D**ecrement **C**ou**NT**er **N**ot **Z**ero'' jumps if the controller's top of counter stack is not zero (before the decrement) and then decrement it.

```
jdcntnz  1223
```

Identical to **`endloop`**, but without the assembler matching the label up with the appropriate **`beginLoop`**.

Kestrel binary instruction format

Array instruction: higher 44 bits for instruction, plus 8 bits for array immediate value.															
FORCE	ALU func	C_{in}	**mp**	le	fi	flag bus	bit shift	res sel	SRAM r	SRAM w	OpB sel	OpA reg	OpC reg	Dest reg	Arry imm
1	5	1	1	1	1	5	4	2	1	1	3	6	6	6	8

95 bit position 52
44

Controller instruction: lower 44 bits.					
Flow control: PC select, return stack, jump	Scratch register and WOR	Data movement and array I/O	Break point	Controller immediate	Diagnostics, array imm.
10	5	5	1	16	7

43 bit position
0