General Lessons : Iteration Spaces

- Computation is organised into a hierarchy of iteration spaces
  - ***Work-item***: granularity of control-flow = *one SIMD lane*
  - ***Warp/Wavefront*** : granularity of scheduling = *one Program Counter*
  - ***Local Workgroup*** : collection of work-items within one processor
  - ***Global Workgroup*** : collection of work-items with shared code

General Lessons : Iteration Spaces

- Computation is organised into a hierarchy of iteration spaces
  - ***Work-item***: granularity of control-flow = *one SIMD lane*
  - ***Warp/Wavefront*** : granularity of scheduling = *one Program Counter*
  - ***Local Workgroup*** : collection of work-items within one processor
  - ***Global Workgroup*** : collection of work-items with shared code

- Need to have an appropriate sizes for each level
  - ***Work-item*** : Startup cost vs amount of work done
    - *If your kernel doesn't contain a loop, how much work can it do?*
  - ***Local group*** : balance registers/thread against threads/block
    - *Want lots of warps ready to run; hide ALU and memory latency*
  - ***Global group*** : Want enough grids to utilise all processors
    - *Balance no. of threads vs startup cost of thread*

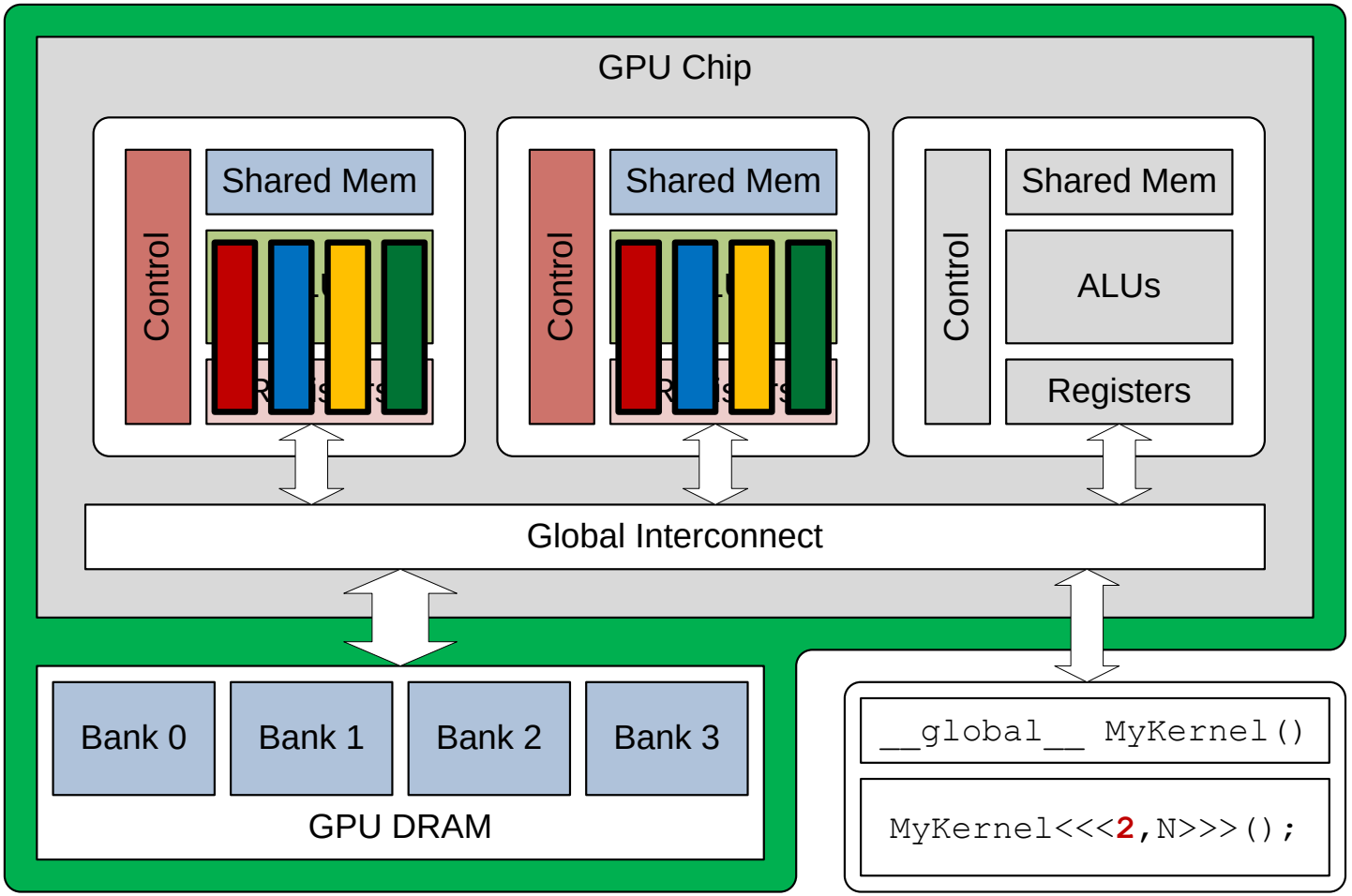
General Lessons : Communication

- ***Registers*** : Local to just one thread
  - Each thread has a unique copy of variables in the kernel
- ***Local Memory*** : Shared within just one block
  - Can be used to communicate between threads
  - Threads within warp should try to read/write non-conflicting banks
- ***Global Memory*** : Shared amongst all work-items in a GPU
  - Threads can communicate with any thread in ***any*** grid
  - Allocated and freed by host using `cl::Buffer`
  - State is maintained between grid executions

- ***Host Memory*** : Local to CPU – "normal" RAM
  - GPU and CPU have different address spaces
  - Use `enqueueRead/Write` to move data between them

Putting it all together

- We want to try and use multiple processors at once
- Each local group executes on one processor: need many groups
- Launch large global groups, will be scheduled on all processors



What is a GPU not good at?

- What pushes against the design parameters?
- **Branching** : divergent control-flow is not cheap
  - What if all threads take a different branch?
- **Small tasks** : things which can't be bundled into blocks
  - If there is only one thread in a block it will be very slow
- **Irregular accesses** : scalar reads/writes to global memory
  - One thread accesses global memory -> all threads in warp stall

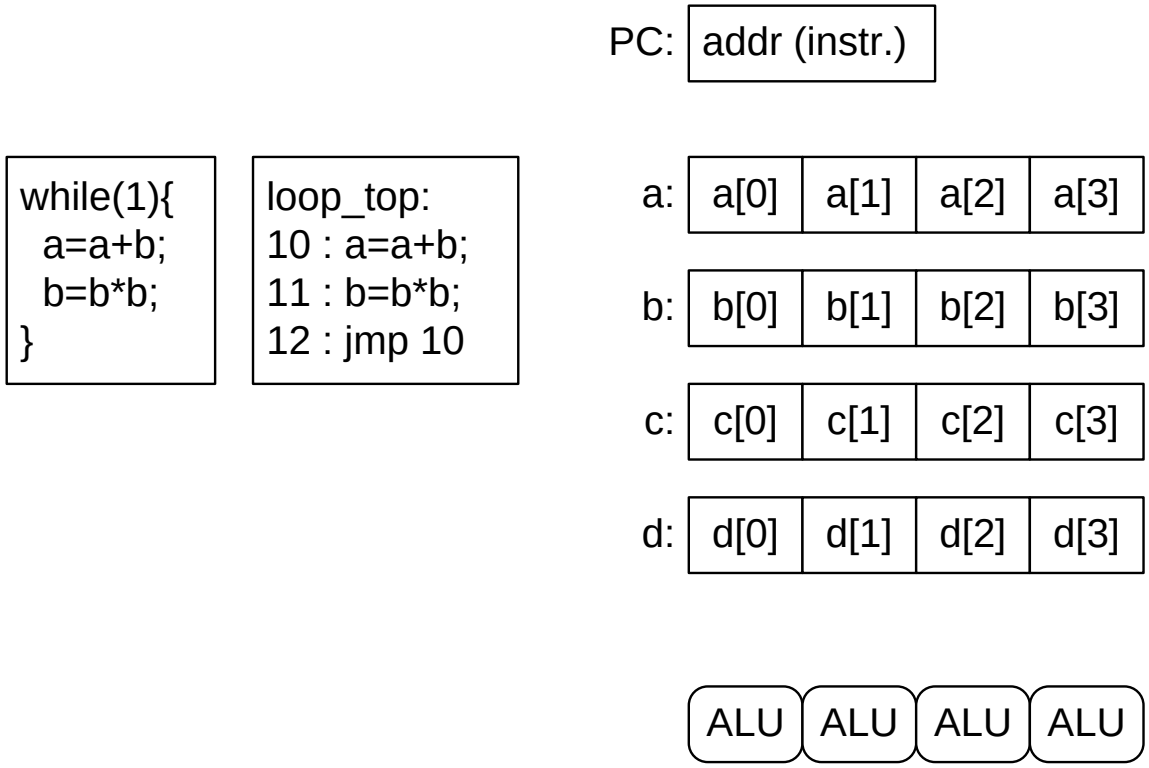
What is a GPU not good at?

- What pushes against the design parameters?

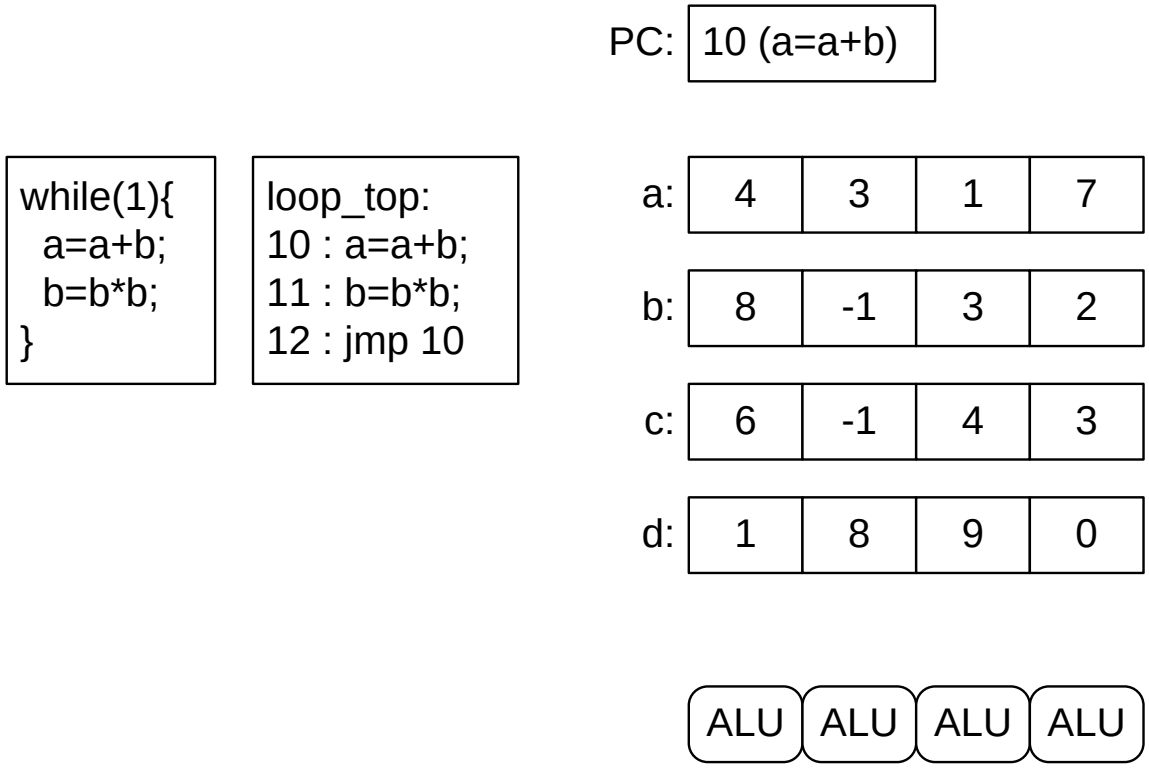
- **Branching** : divergent control-flow is not cheap
  - What if all threads take a different branch?

- **Small tasks** : things which can't be bundled into blocks
  - If there is only one thread in a block it will be very slow

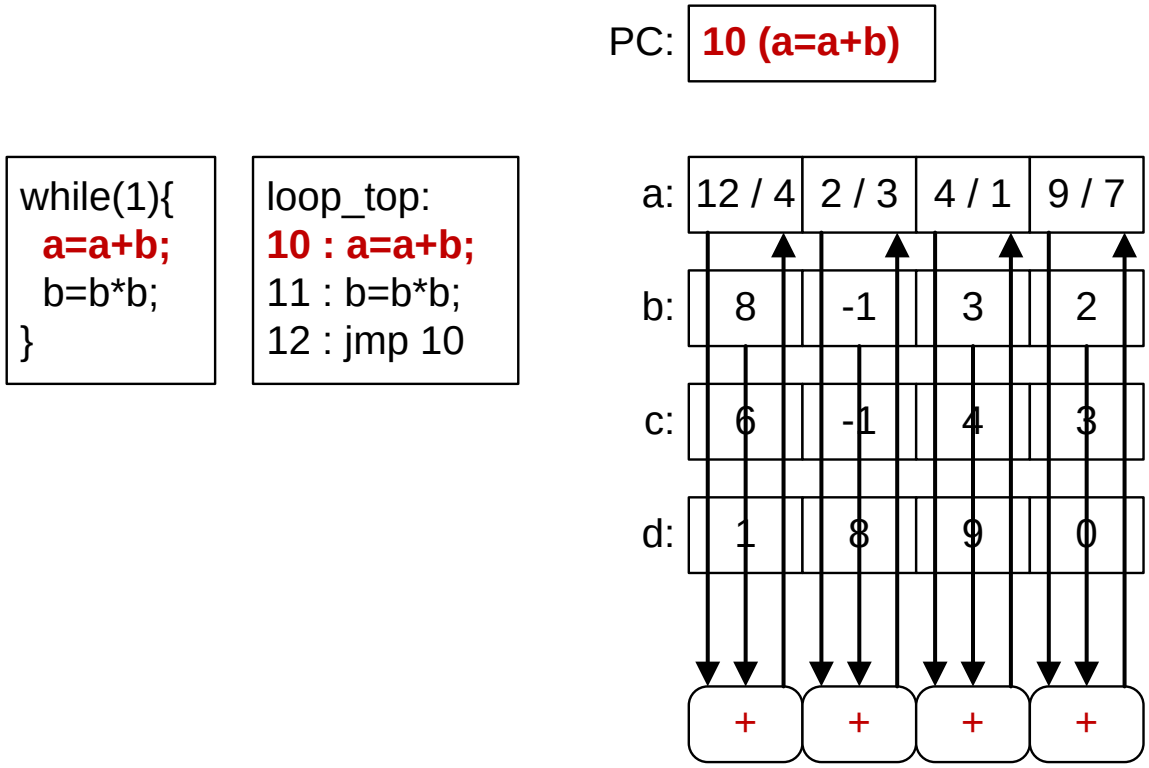
- **Irregular accesses** : scalar reads/writes to global memory
  - One thread accesses global memory -> all threads in warp stall



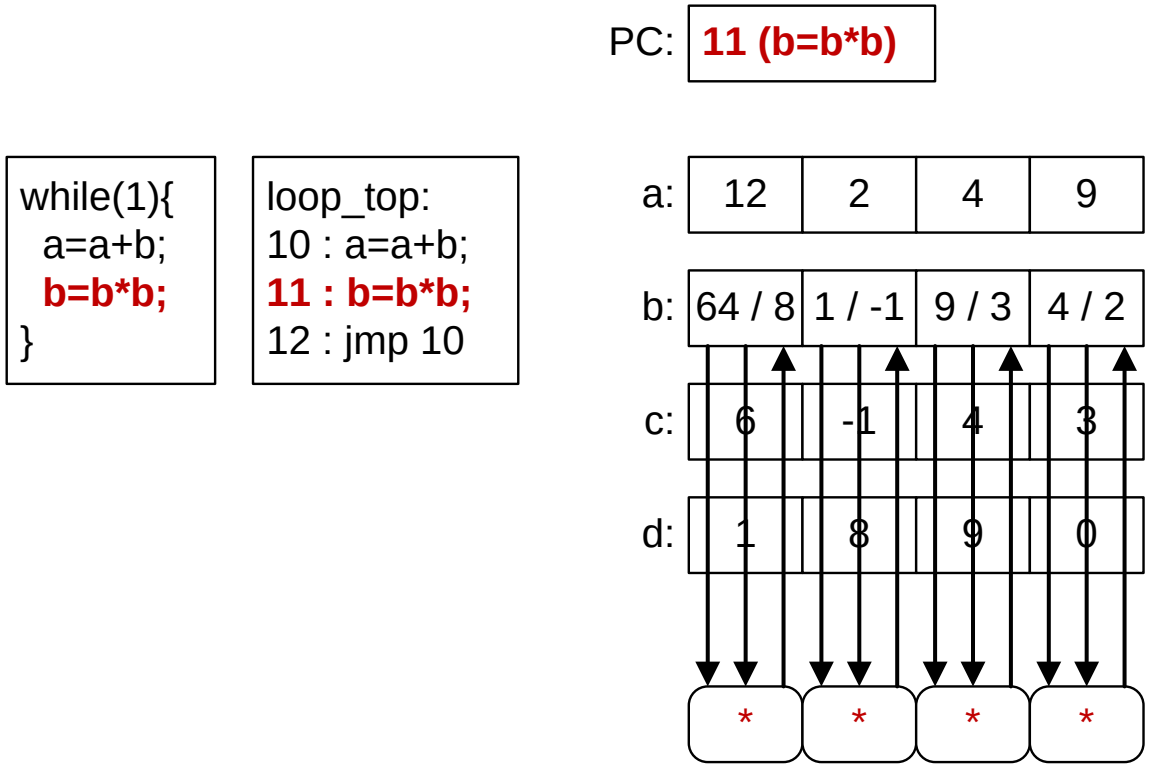
- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU



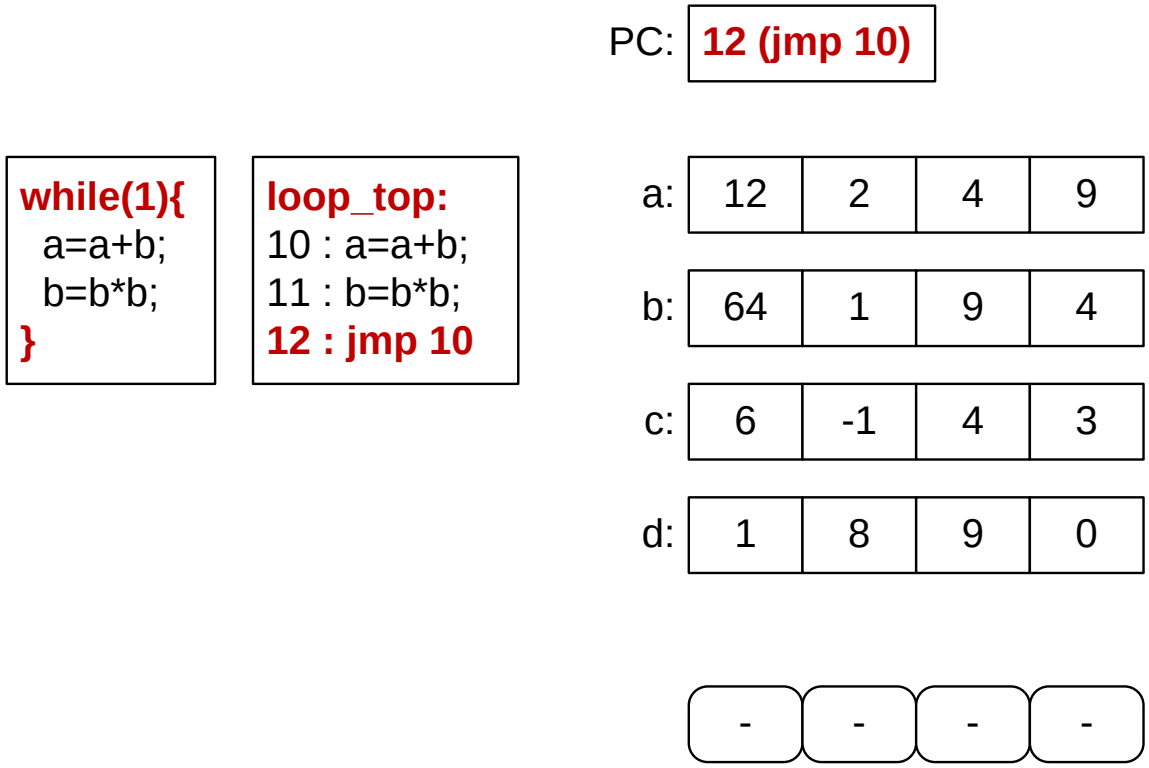
- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU



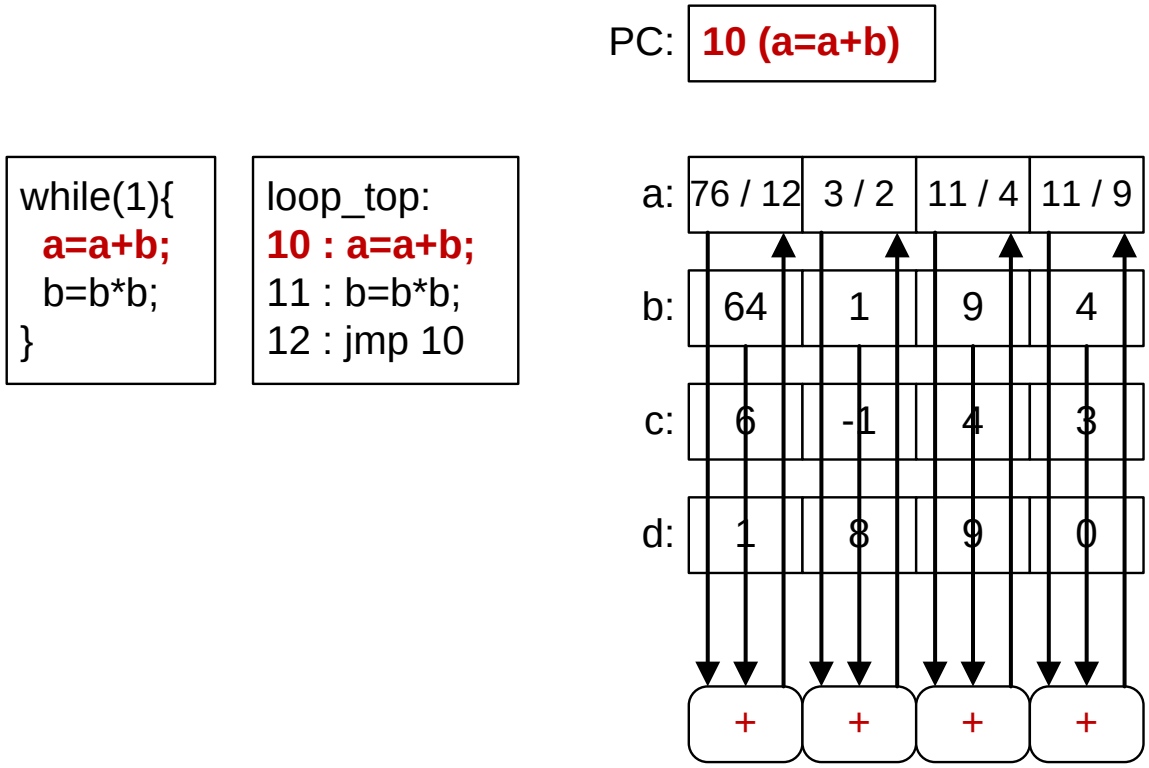
- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU
- Registers are modified using ALUs
  - Using syntax Y / X to mean: “X” at end of cycle, “Y” at start



- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU
- Registers are modified using ALUs
  - Using syntax Y / X to mean: “X” at end of cycle, “Y” at start



- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU
- Registers are modified using ALUs
  - Using syntax Y / X to mean: “X” at end of cycle, “Y” at start



- Plain SIMD : each register expands to multiple lanes
  - Each lane is associated with one ALU
- Registers are modified using ALUs
  - Using syntax Y / X to mean: “X” at end of cycle, “Y” at start

Branches

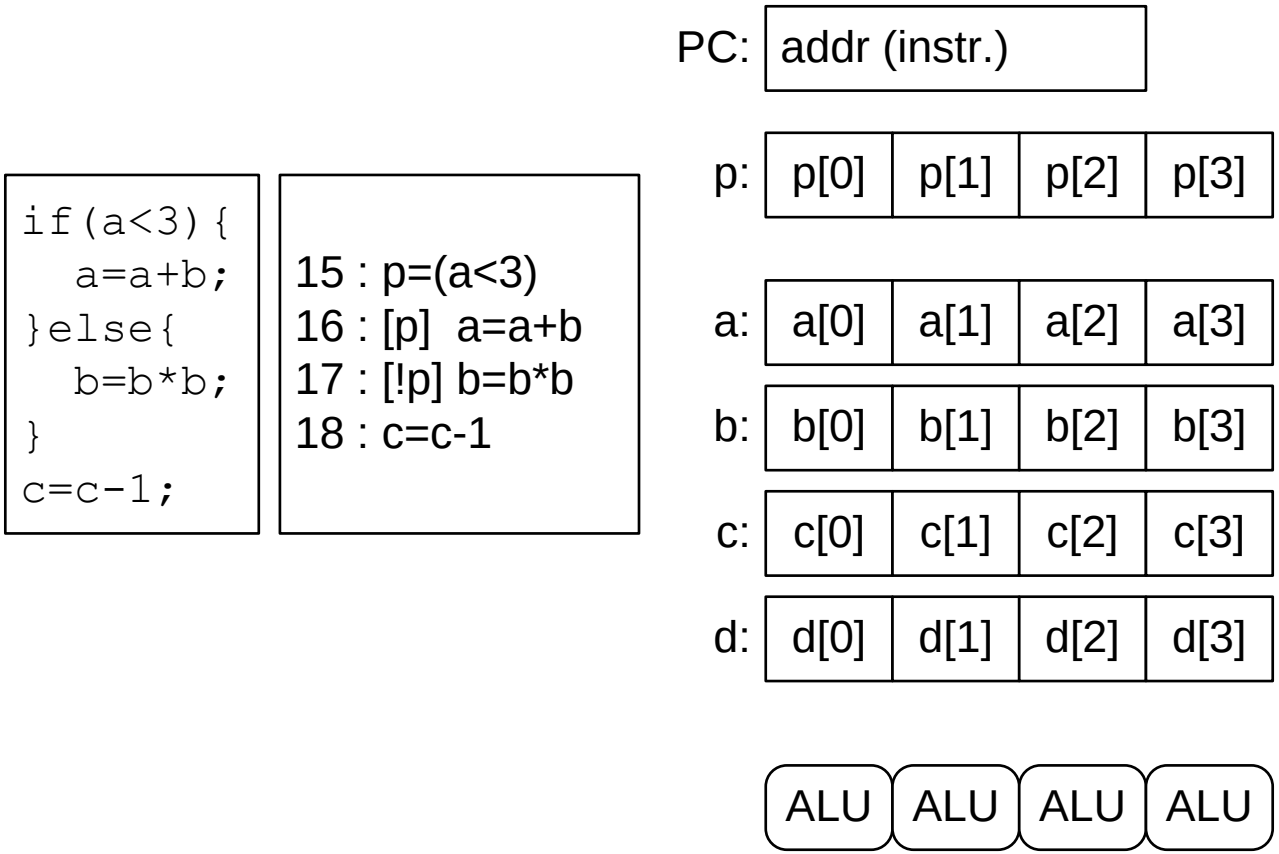
- What happens when there are branches within lanes?
  - Each work-item is a thread of control
  - Can branch or loop however it wants
- Need to apply operations to just a sub-set of lanes

Branches

- What happens when there are branches within lanes?
  - Each work-item is a thread of control
  - Can branch or loop however it wants
- Need to apply operations to just a sub-set of lanes

- Use predicate register to control ALUs
  - One bit per lane
  - Can be modified with comparison instructions
  - Standard instructions can be guarded with predicate

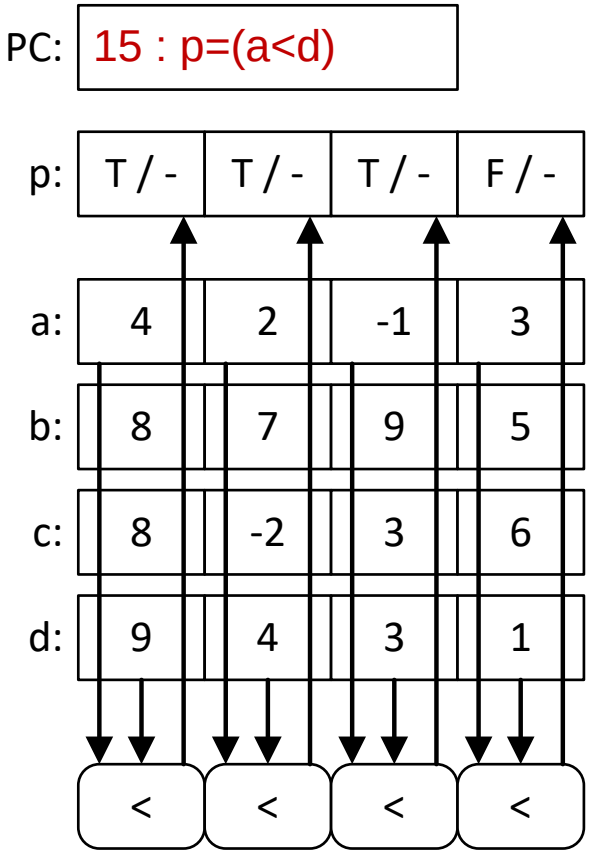
- Somewhat similar to ARM conditional execution



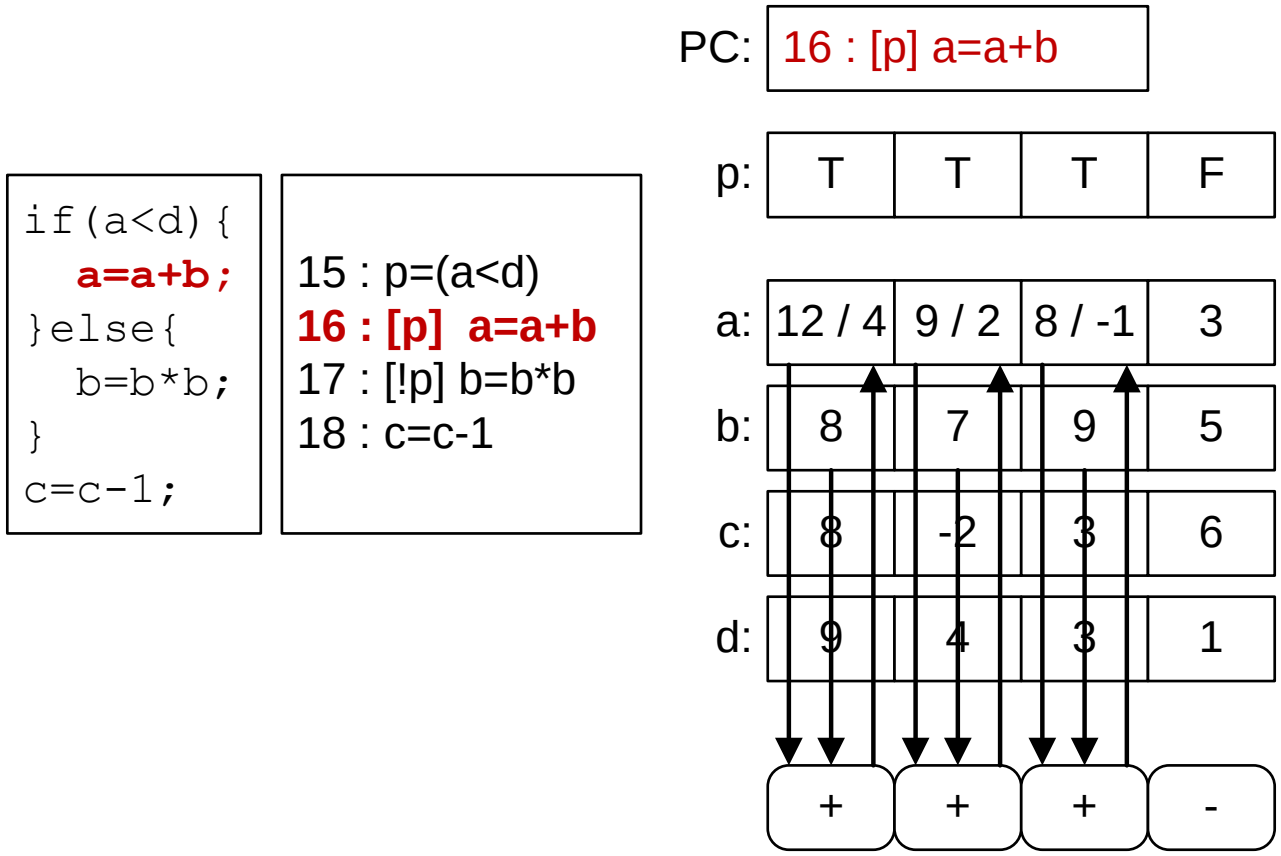
- New 1-bit predicate register per lane

```
if(a<d){
  a=a+b;
}else{
  b=b*b;
}
c=c-1;
```

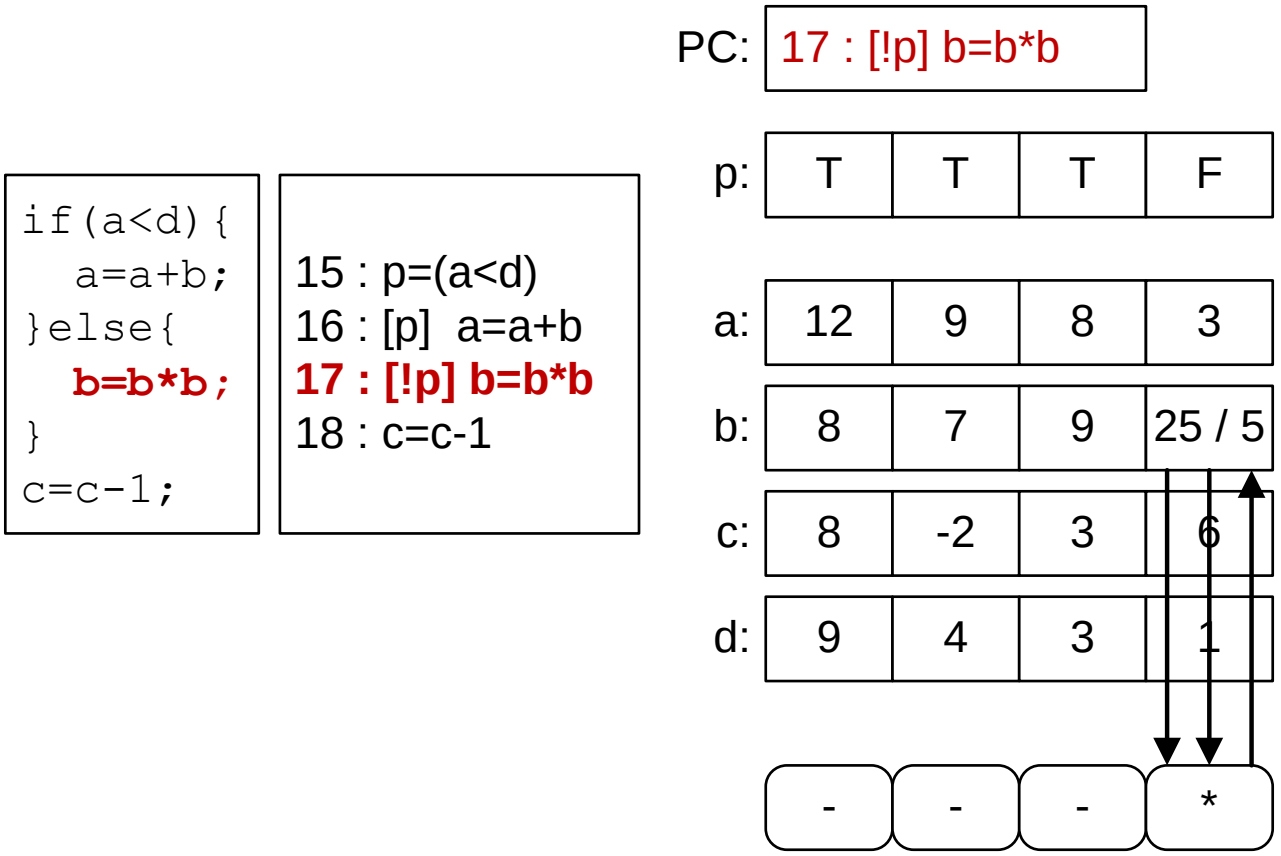
15 : p=(a<d)
16 : [p] a=a+b
17 : [!p] b=b*b
18 : c=c-1



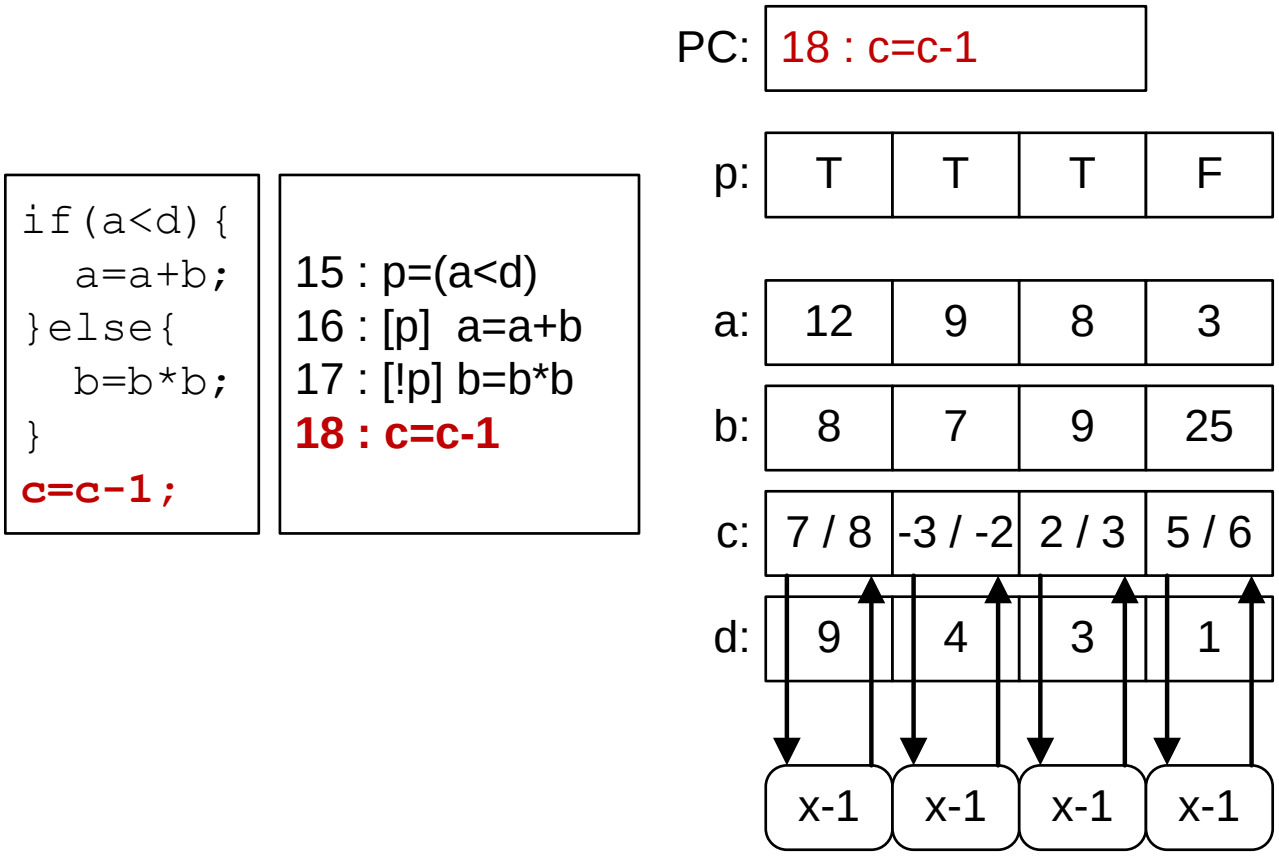
- New 1-bit predicate register per lane



- New 1-bit predicate register per lane



- New 1-bit predicate register per lane



- New 1-bit predicate register per lane

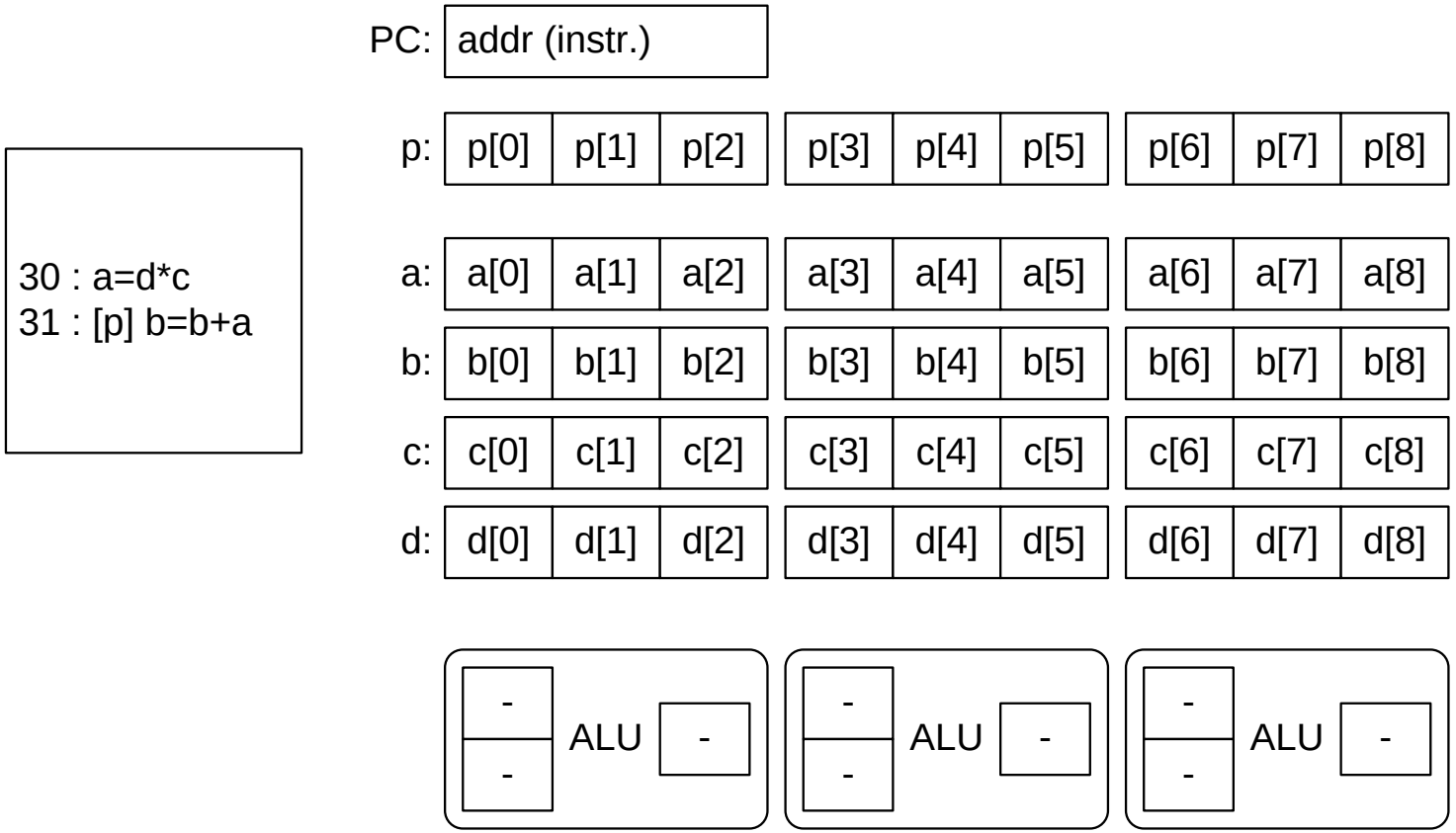
Improving Efficiency

- ALUs are big, typically much bigger than registers
- Combinatorial floating-point ALUs would be very slow

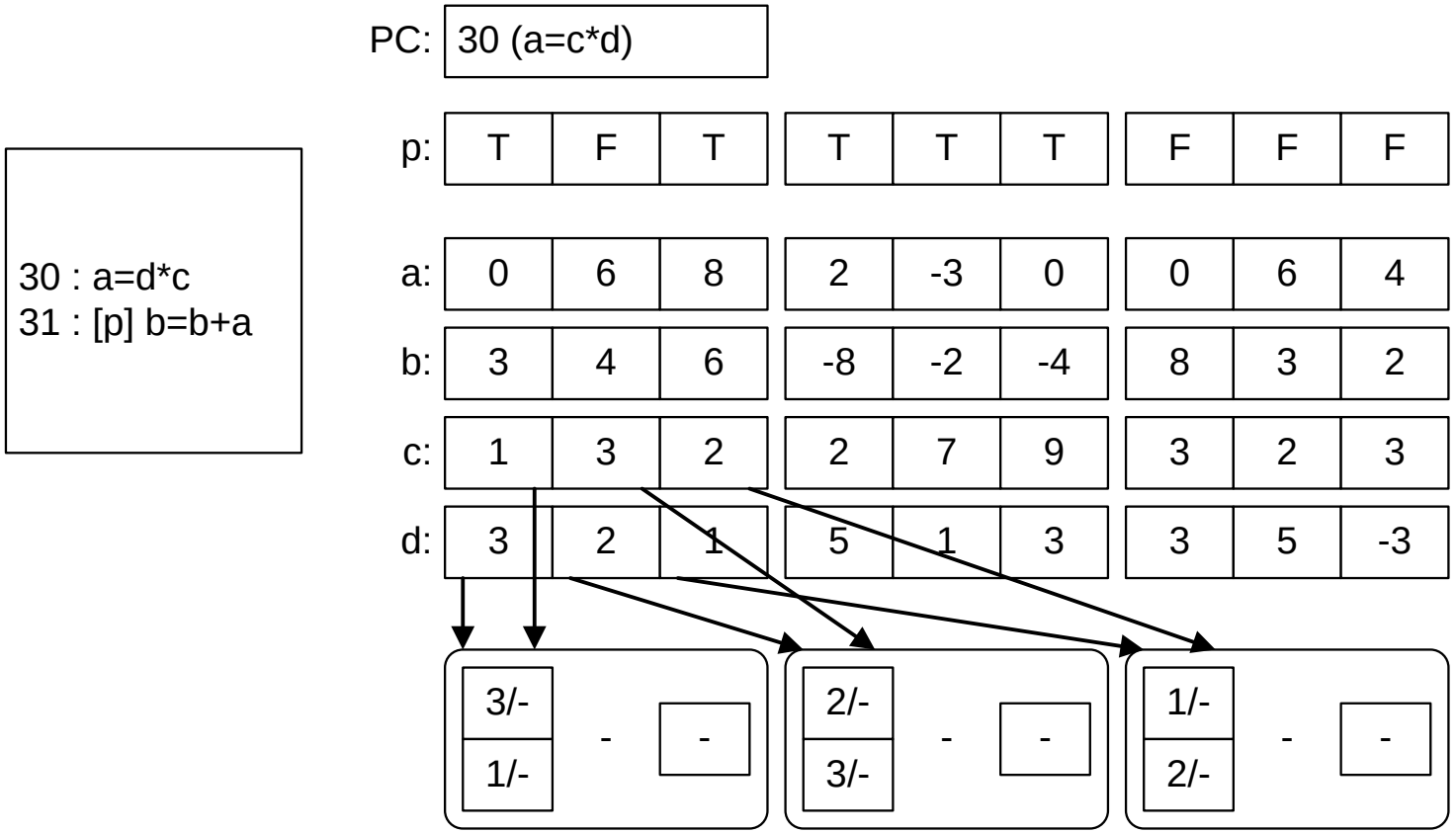
Improving Efficiency

- ALUs are big, typically much bigger than registers
- Combinatorial floating-point ALUs would be very slow

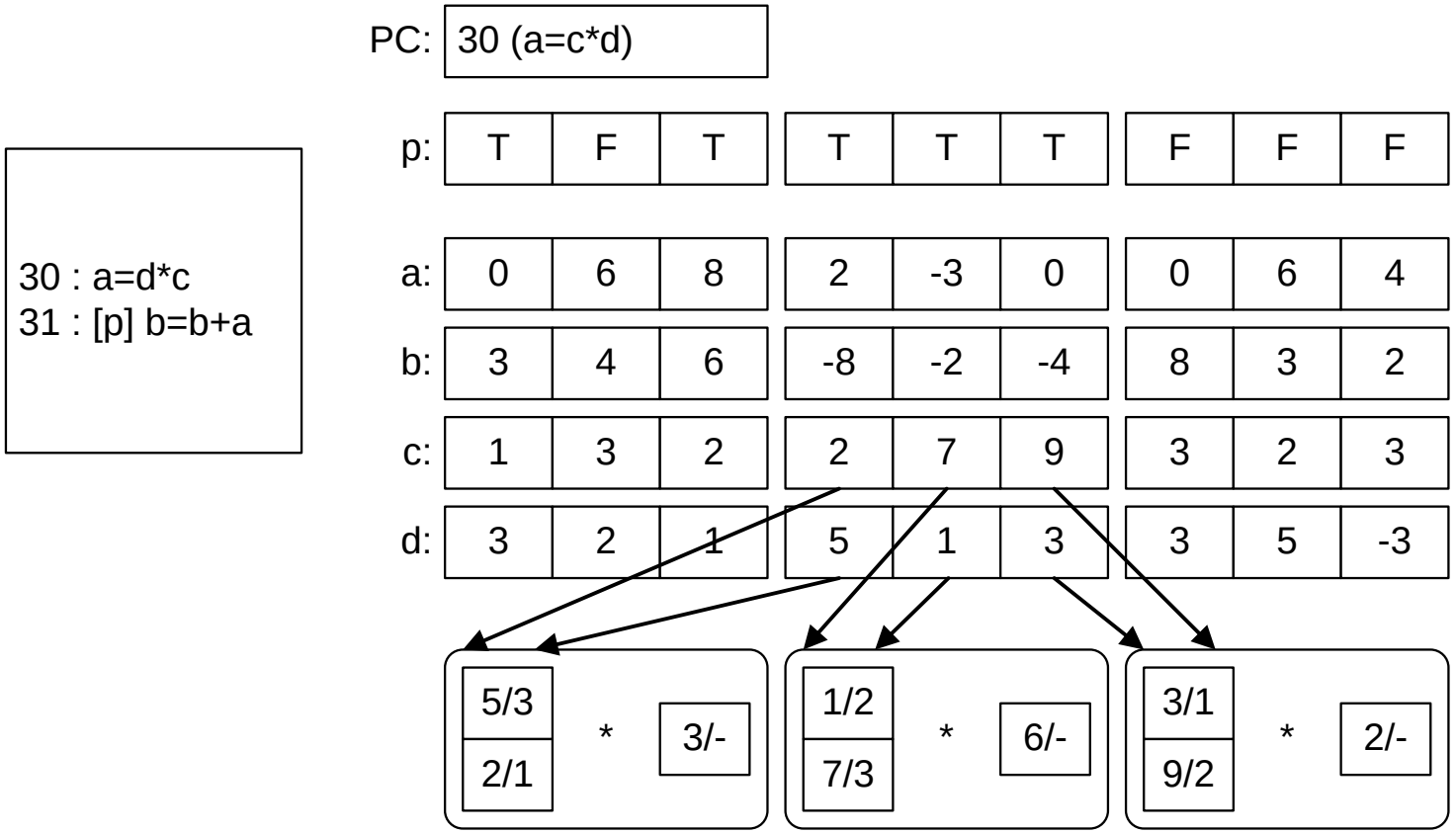
- Solution: time-multiplex register lanes on to ALUs
  - Create n actual ALU lanes
  - Create k*n lanes of storage in each register
  - Execute operations over k successive cycle



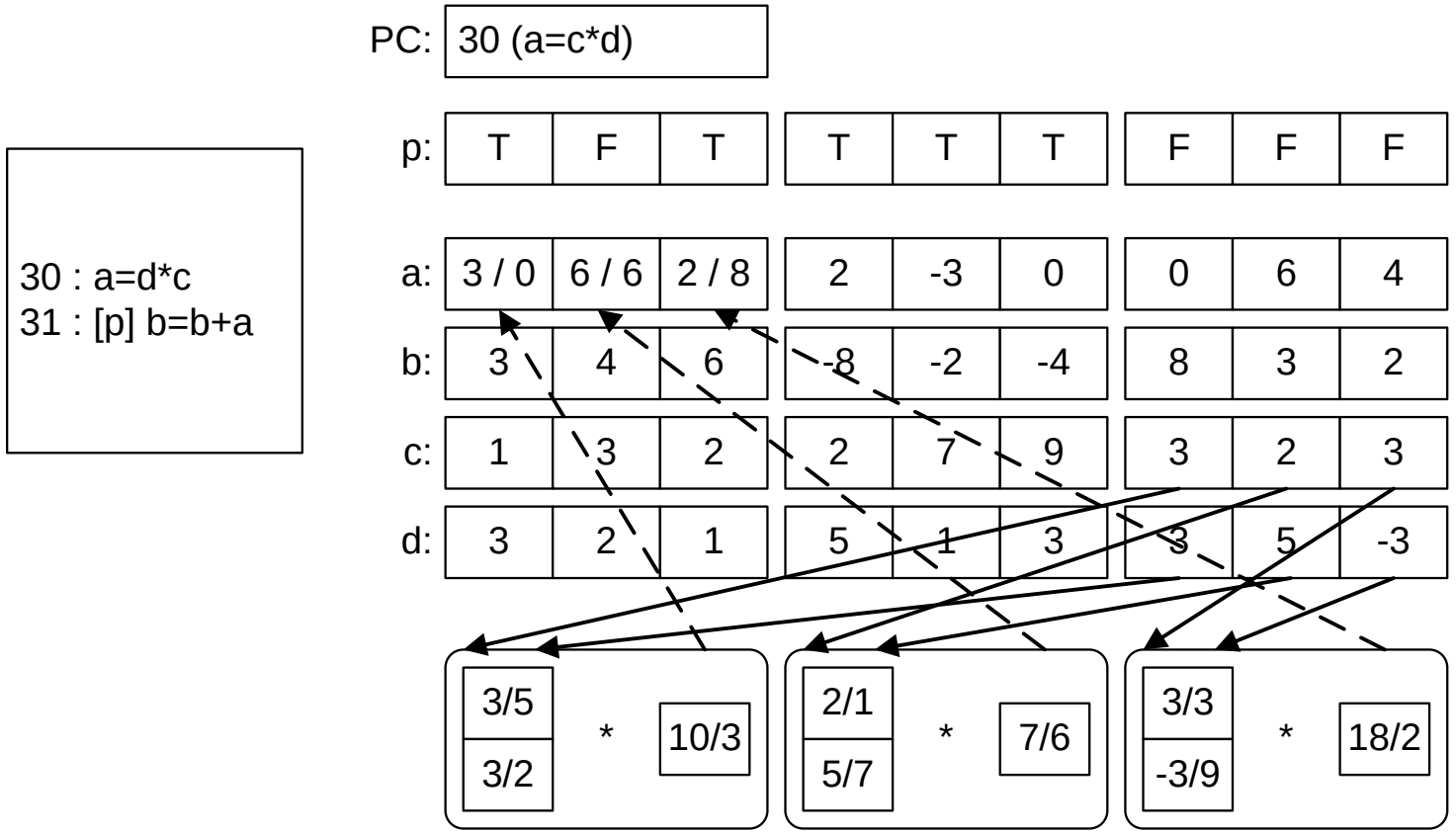
- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



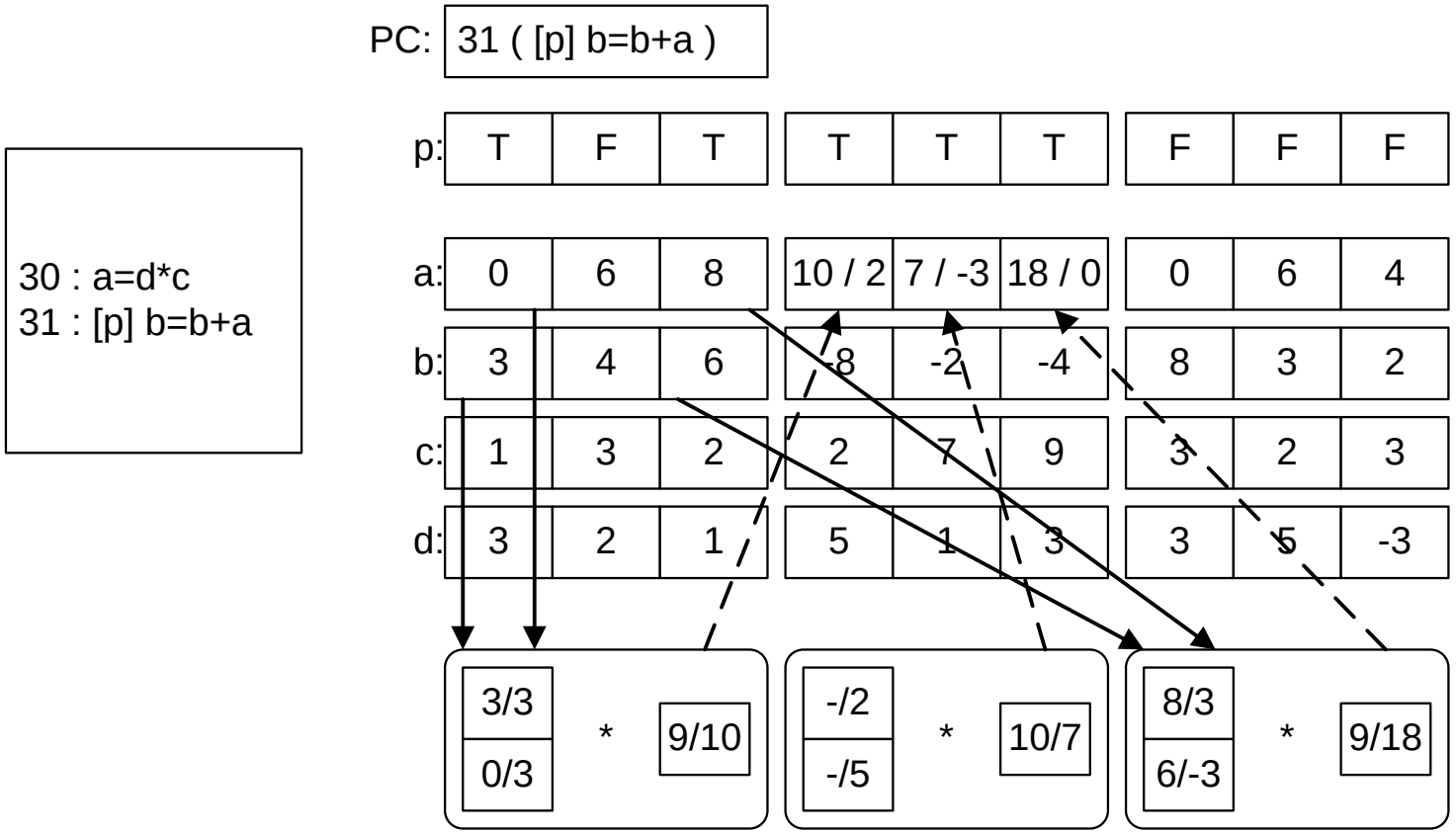
- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



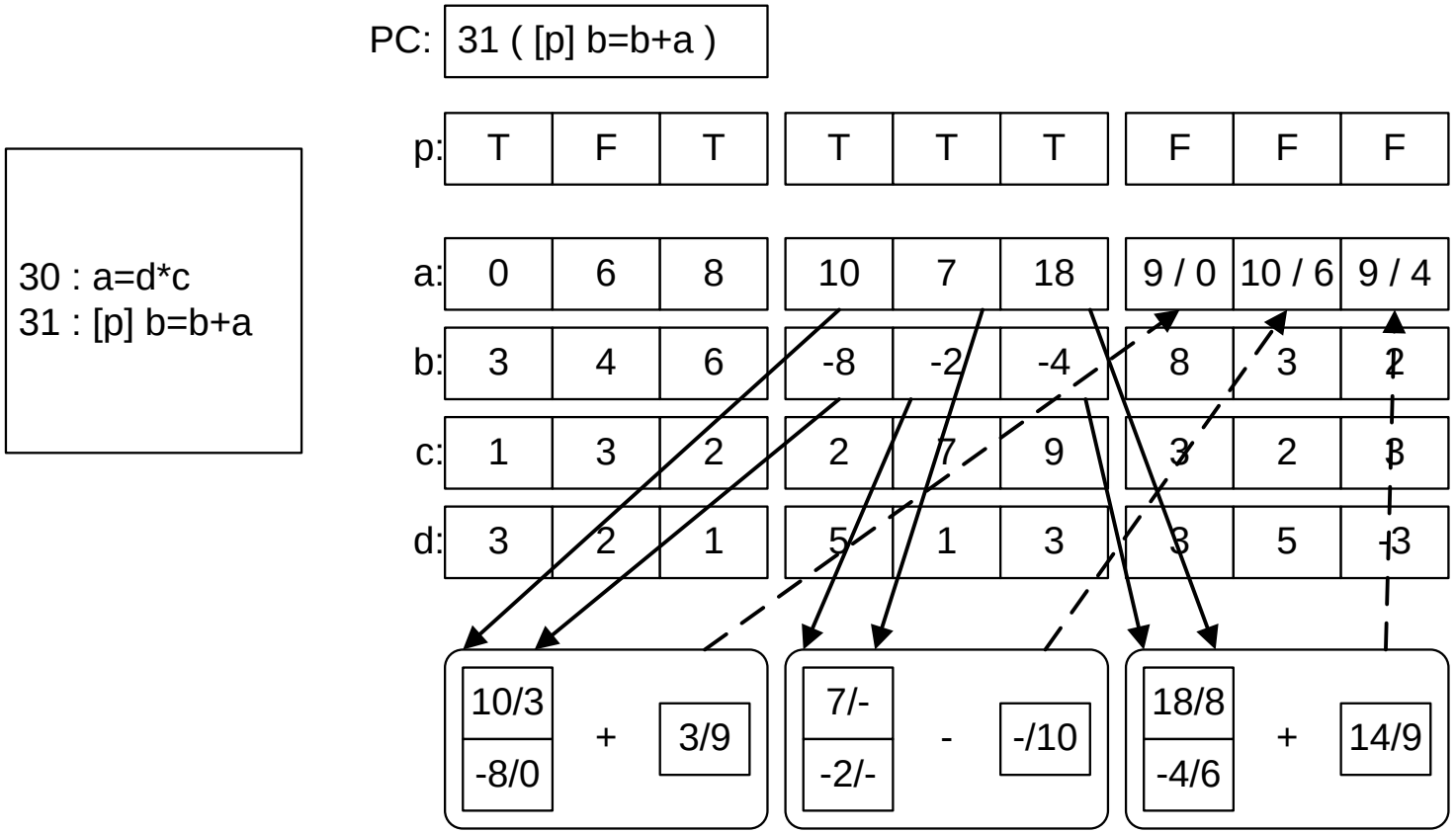
- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



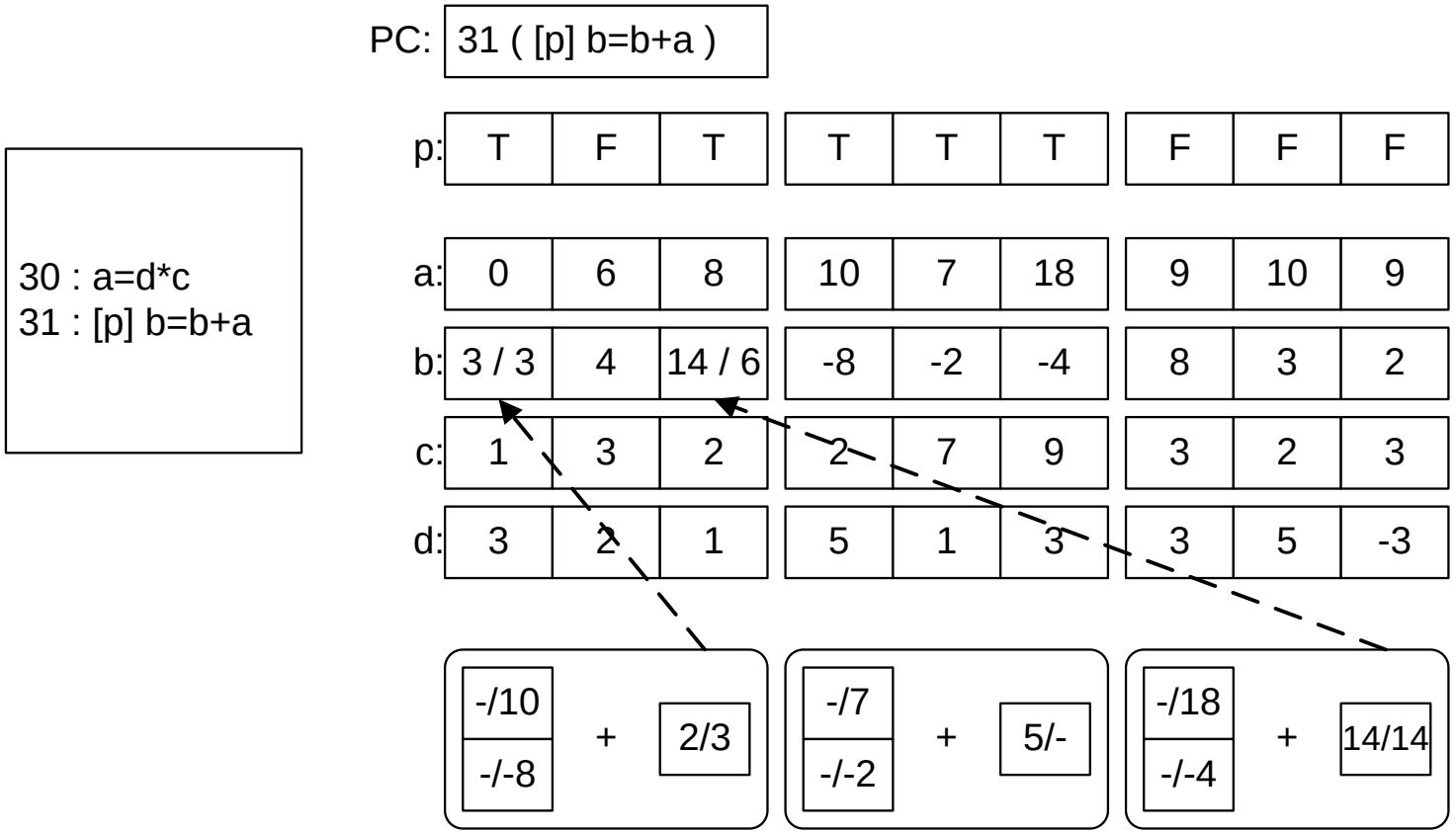
- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



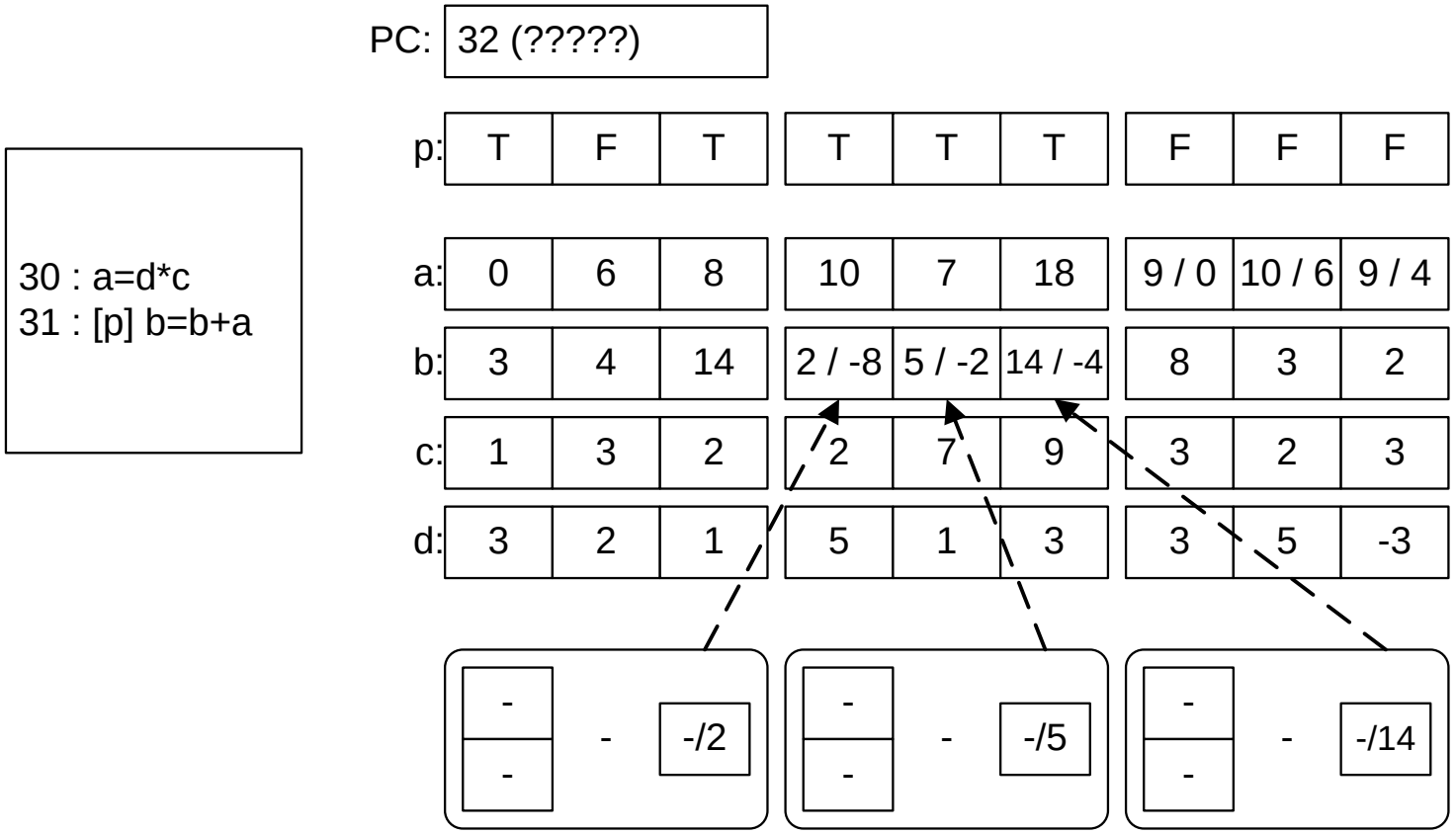
- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through



- Three actual ALU lanes, with nine lanes in registers
  - ALUs are pipelined – takes two cycles to propagate through

Some informal terminology

- Each lane is the OpenCL work-item (or thread)
  - Smallest unit of scheduling within GPU
- Group of ALU lanes: Warp (NVidia) or Wavefront (AMD)
  - Warp size is the number of ALU operations performed in parallel
- Group of register lanes: ~ local work-group (OpenCL)
  - User chooses their block size, i.e. how many threads / local group
  - Block size is usually small multiple of warp size
    - Want at least 4-6 warps to hide ALU latency

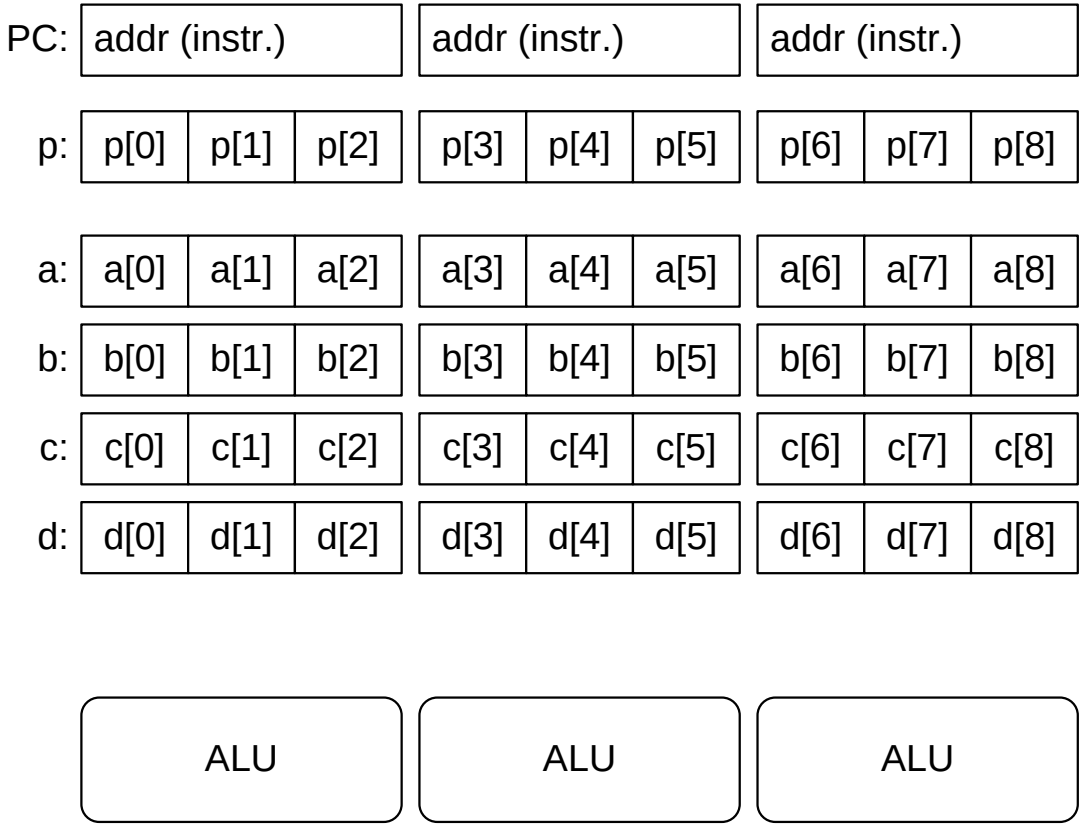
(no real agreement on what these things are called in hardware)

Warp Divergence

- It’s useful to allow threads to execute different paths
  - All threads have the same instruction stream
  - Threads don’t have to follow the same path

- General principle of diverge and re-combine
  - Each thread checks a condition during a branch instruction
  - If threads don’t agree on branch, execute one path then the other

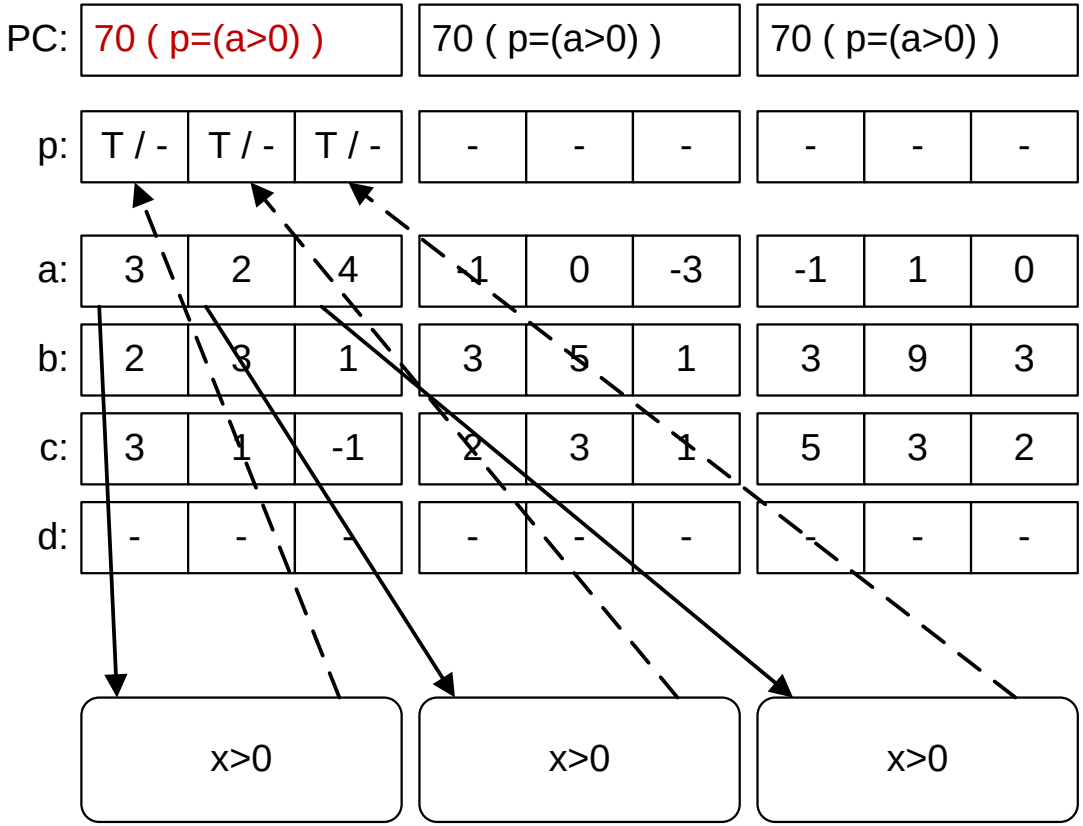
```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



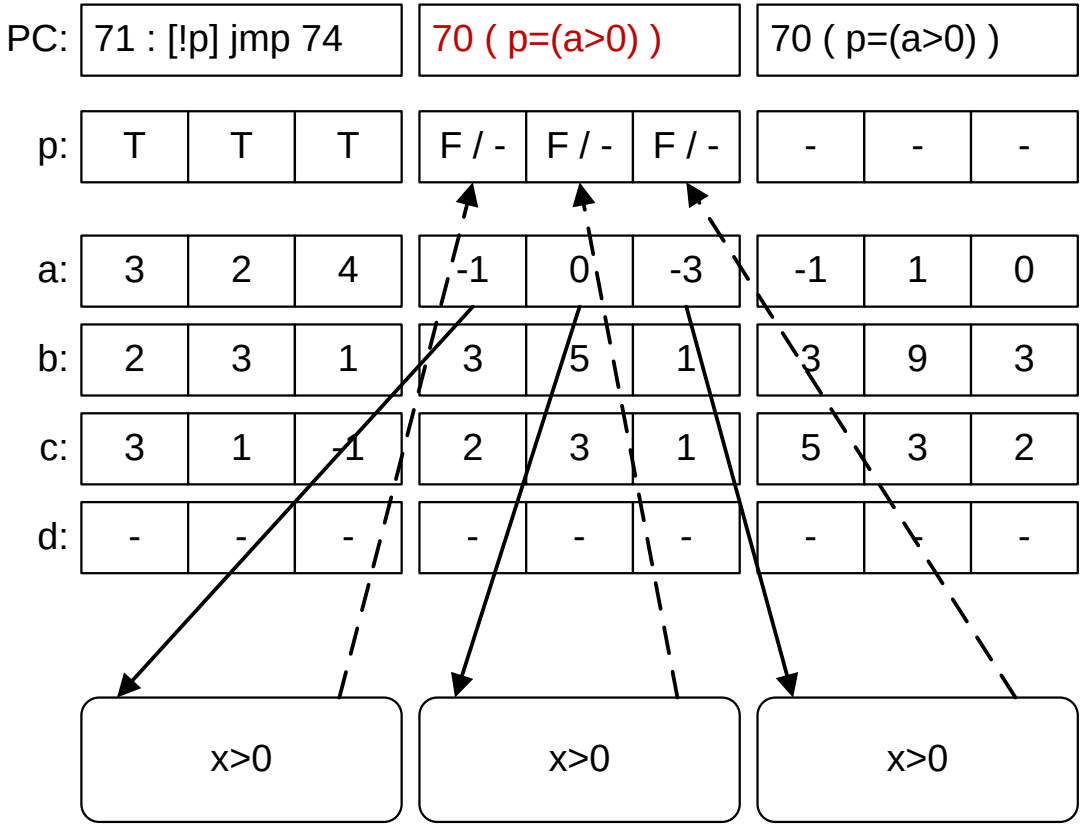
```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```

PC:	70 (p=(a>0))			70 (p=(a>0))			70 (p=(a>0))		
p:	-	-	-	-	-	-	-	-	-
a:	3	2	4	-1	0	-3	-1	1	0
b:	2	3	1	3	5	1	3	9	3
c:	3	1	-1	2	3	1	5	3	2
d:	-	-	-	-	-	-	-	-	-
	ALU			ALU			ALU		

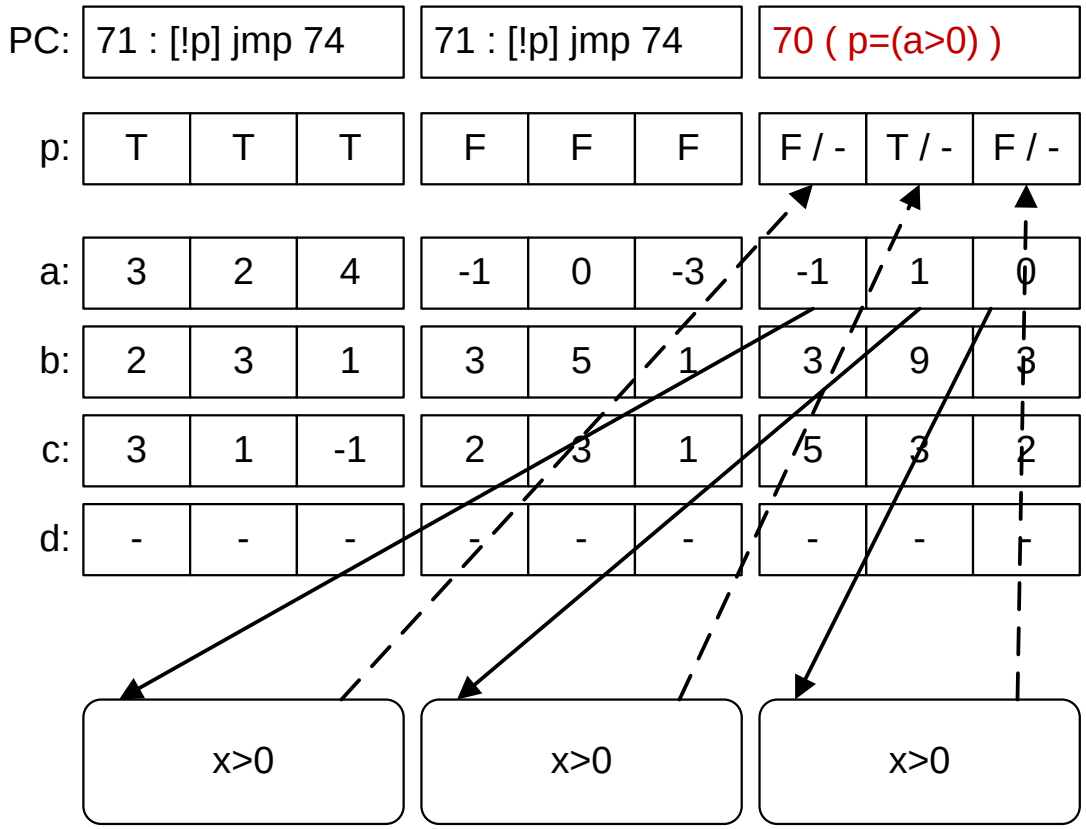
```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```

PC:	71 : [!p] jmp 74			71 : [!p] jmp 74			71 : [!p] jmp 74		
p:	T	T	T	F	F	F	F	T	F
a:	3	2	4	-1	0	-3	-1	1	0
b:	2	3	1	3	5	1	3	9	3
c:	3	1	-1	2	3	1	5	3	2
d:	-	-	-	-	-	-	-	-	-
	-			-			-		

```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```

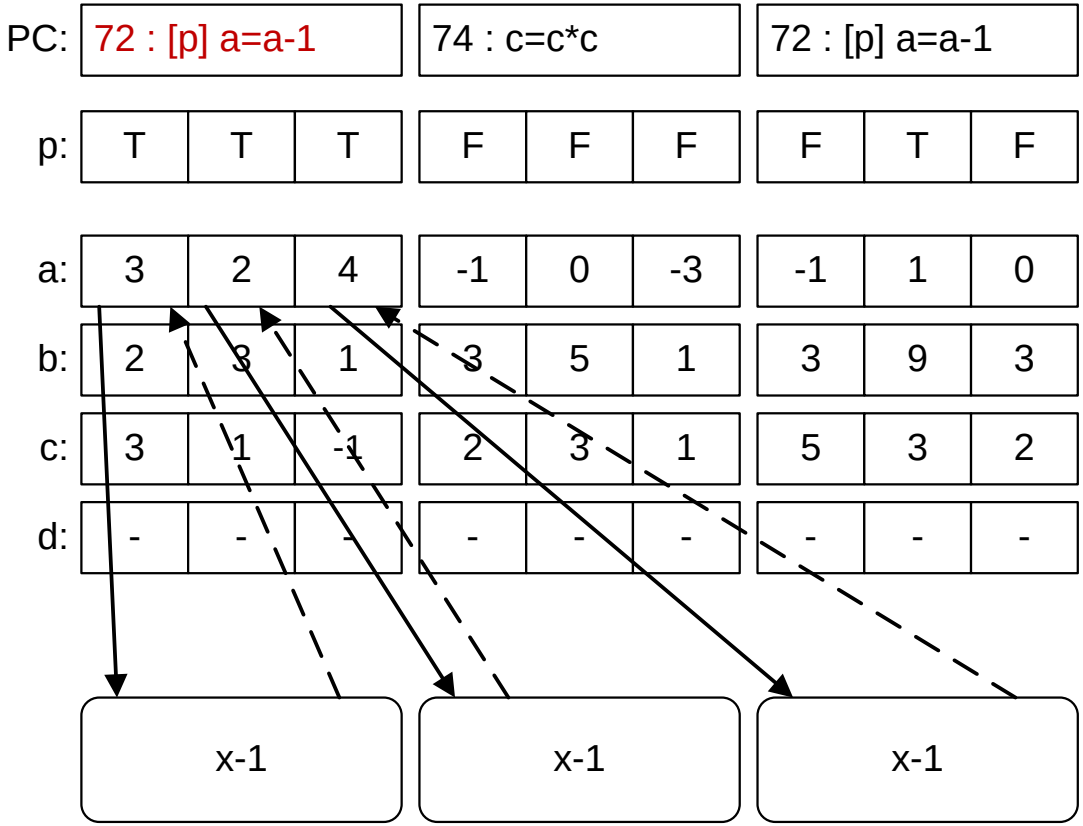
PC:	72 : [p] a=a-1			71 : [!p] jmp 74			71 : [!p] jmp 74		
p:	T	T	T	F	F	F	F	T	F
a:	3	2	4	-1	0	-3	-1	1	0
b:	2	3	1	3	5	1	3	9	3
c:	3	1	-1	2	3	1	5	3	2
d:	-	-	-	-	-	-	-	-	-
	-			-			-		

```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```

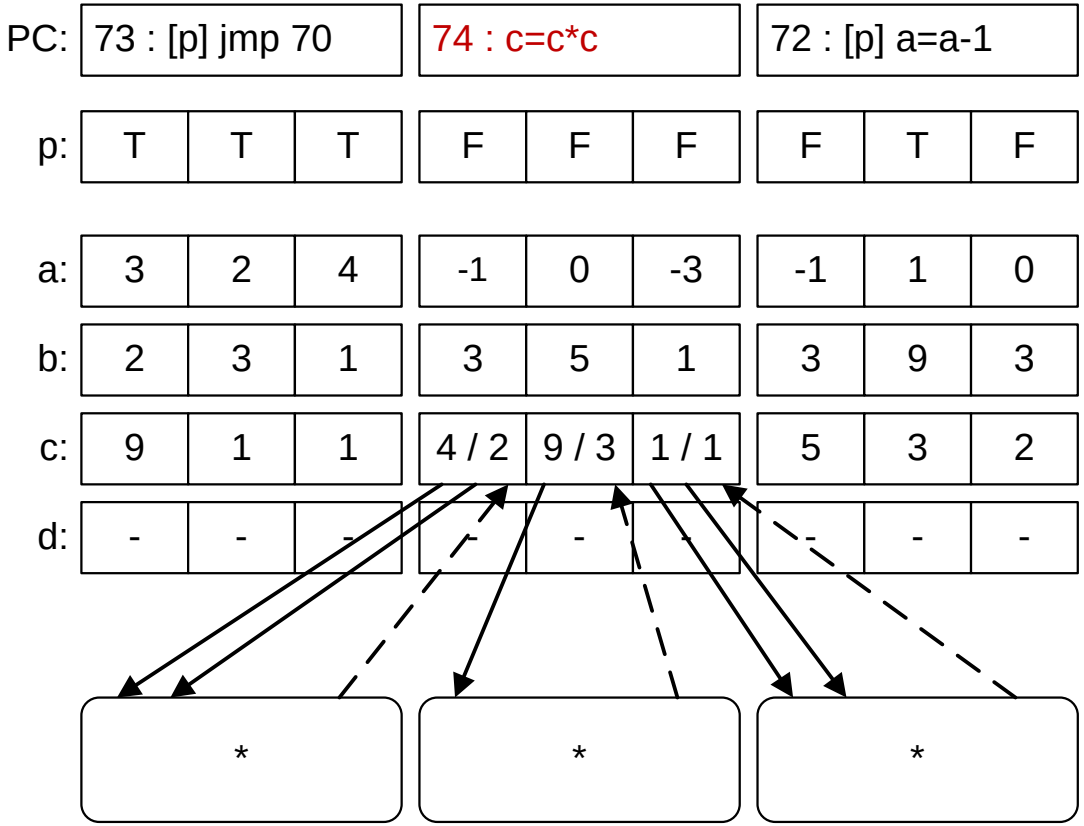
PC:	72 : [p] a=a-1			74 : c=c*c			71 : [!p] jmp 74		
p:	T	T	T	F	F	F	F	T	F
a:	3	2	4	-1	0	-3	-1	1	0
b:	2	3	1	3	5	1	3	9	3
c:	3	1	-1	2	3	1	5	3	2
d:	-	-	-	-	-	-	-	-	-

-	-	-

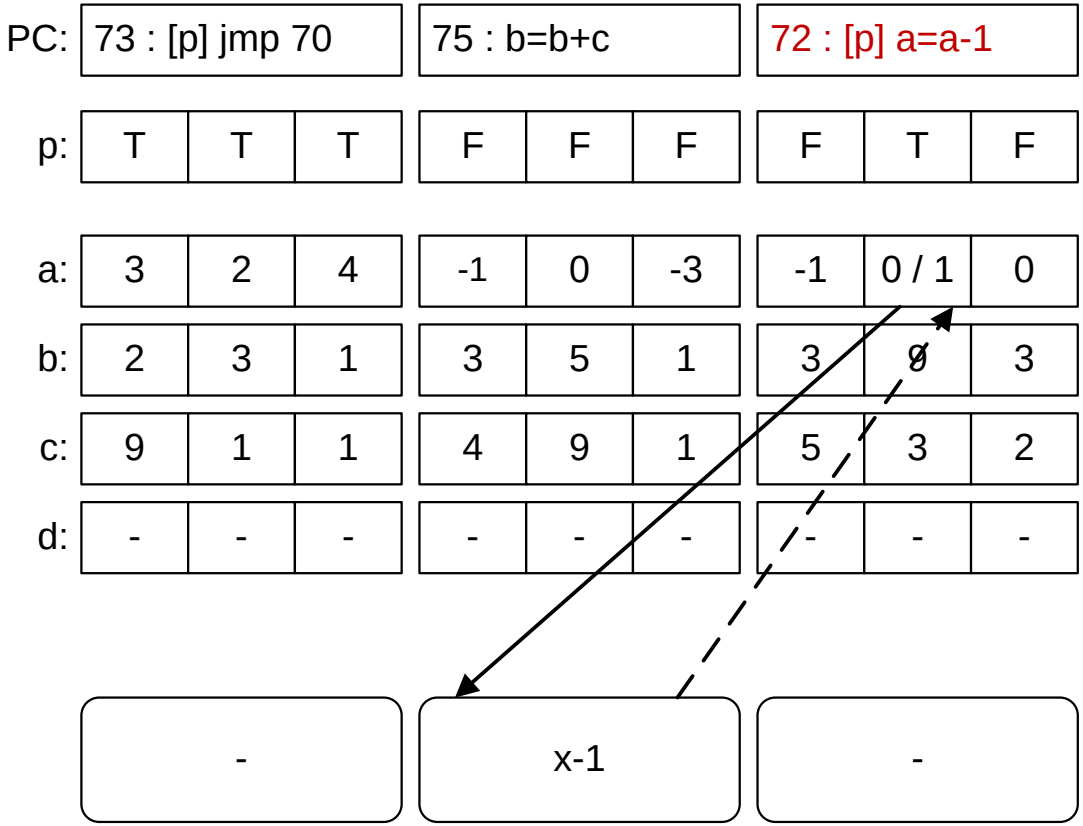
```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



```
70 : p = (a>0)
71 : [!p] jmp 74
72 : [p] a=a-1
73 : [p] jmp 70
74 : c=c*c
75 : b=b+c
```



Reconvergence: synchronisation

- Threads often need to synchronise execution
  - Co-ordinating reads/writes to memory
  - Communicating and voting on progress

- Use a `barrier()` call in OpenCL (language builtin)
  - All threads share a single instruction stream
  - Block threads until all threads in block reach the same instruction

- Need to make sure that ***all*** threads will execute synchronisation
  - What happens if there is a barrier inside a branch?

Thinking about divergence

- The GPU model assumes a dense regular iteration space
  - Works well if all threads do the same amount of work
  - What if some threads do lots of work, but some do none?

```
__kernel void MyKernel()
{
    // Prob. p of being taken
    if(condition(get_id(0))){
        // Takes time t
        DoHugeAmountsOfWork();
    }
}
```

Thinking about divergence

- The GPU model assumes a dense regular iteration space
  - Works well if all threads do the same amount of work
  - What if some threads do lots of work, but some do none?
- Assume a warp size of K
  - Prob. of ***no*** threads taking branch is $(1-p)^K$
  - Prob. of ***any*** threads = $1-(1-p)^K$

```
__kernel void MyKernel()
{
    // Prob. p of being taken
    if(condition(get_id(0))){
        // Takes time t
        DoHugeAmountsOfWork();
    }
}
```

Thinking about divergence

- The GPU model assumes a dense regular iteration space
  - Works well if all threads do the same amount of work
  - What if some threads do lots of work, but some do none?
- Assume a warp size of K
  - Prob. of ***no*** threads taking branch is $(1-p)^K$
  - Prob. of ***any*** threads = $1-(1-p)^K$
- Execution time ~ $t(1-(1-p)^K)$

```
__kernel void MyKernel()
{
    // Prob. p of being taken
    if(condition(get_id(0))){
        // Takes time t
        DoHugeAmountsOfWork();
    }
}
```

Thinking about divergence

- The GPU model assumes a dense regular iteration space
  - Works well if all threads do the same amount of work
  - What if some threads do lots of work, but some do none?
- Assume a warp size of K
  - Prob. of ***no*** threads taking branch is $(1-p)^K$
  - Prob. of ***any*** threads = $1-(1-p)^K$
- Execution time ~ $t(1-(1-p)^K)$
- What is the expected execution time of a local group?

```
__kernel void MyKernel()
{
    // Prob. p of being taken
    if(condition(get_id(0))){
        // Takes time t
        DoHugeAmountsOfWork();
    }
}
```

Estimate execution time

- We have K threads/warp, N threads/local, M threads/global
  - Total threads = M
  - Warps / local = (N+K-1) div K
  - Warps / global = (M+K-1) div K

Estimate execution time

- We have K threads/warp, N threads/local, M threads/global
  - Total threads = M
  - Warps / local = (N+K-1) div K
  - Warps / global = (M+K-1) div K
- Time of upper hierarchy is max of lower hierarchy
  - *any* active thread will keep a warp active
  - *any* active warp will keep a local group active
  - *any* active local group will keep a global group active

Estimate execution time

- We have K threads/warp, N threads/local, M threads/global
  - Total threads = M
  - Warps / local = (N+K-1) div K
  - Warps / global = (M+K-1) div K
- Time of upper hierarchy is max of lower hierarchy
  - *any* active thread will keep a warp active
  - *any* active warp will keep a local group active
  - *any* active local group will keep a global group active
- Try to work out the worst-case thread execution time
  - ***Excellent*** : all threads follow the same instruction path
  - ***Good*** : all threads in a local group perform the same instructions
  - ***Ok*** : all threads in a warp take the same branches
  - ***Bad*** : each thread takes different path; divergence everywhere