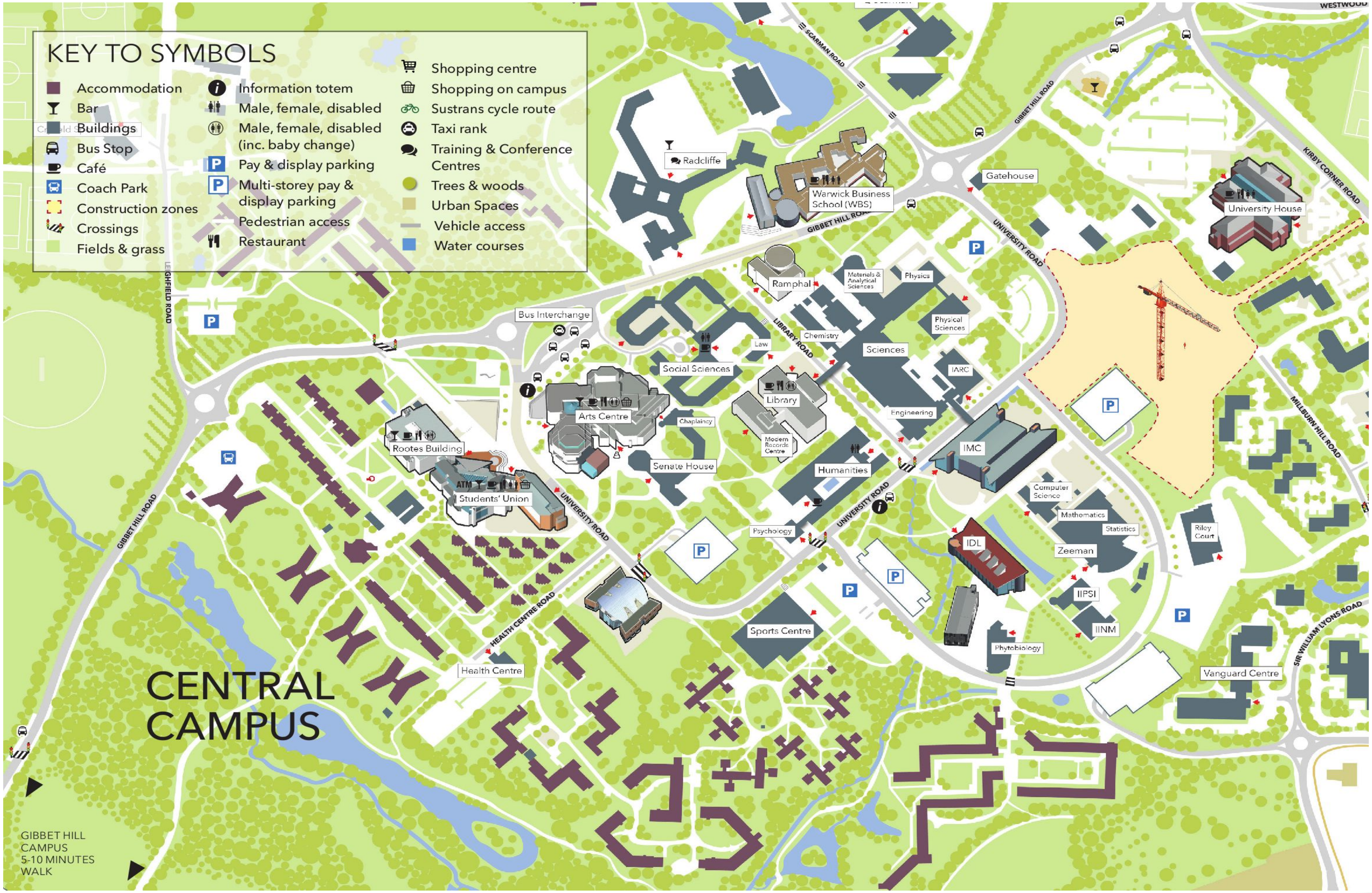
Computing in Space

David Packwood
dpackwood@maxeler.com

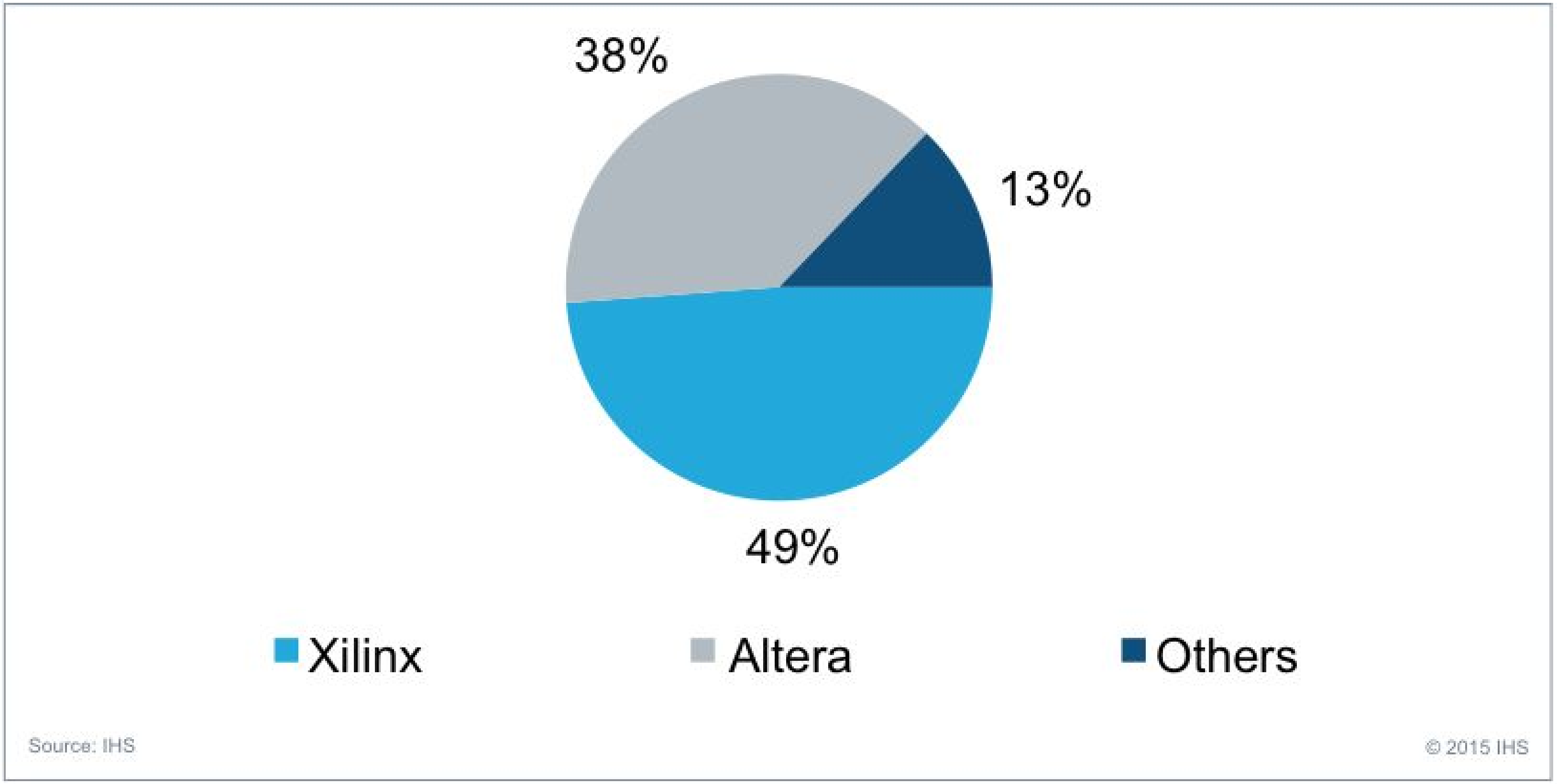
Introduction

- A CPU is effectively a multi-purpose device, it runs operating systems, web browsers, scientific computation and many more.
- Field Programmable Gate Arrays (FPGAs) are a type of computer chip which is repeatedly reconfigurable.
- An FPGA is essentially a large array of low level logical units which can be wired together to form a configuration (called a bitstream).
- Each configuration is designed for a specific task.
- Loosely, because the FPGA can be configured for a specific task it may be able to solve that task much more efficiently than a CPU.

Familiar Visual Aid

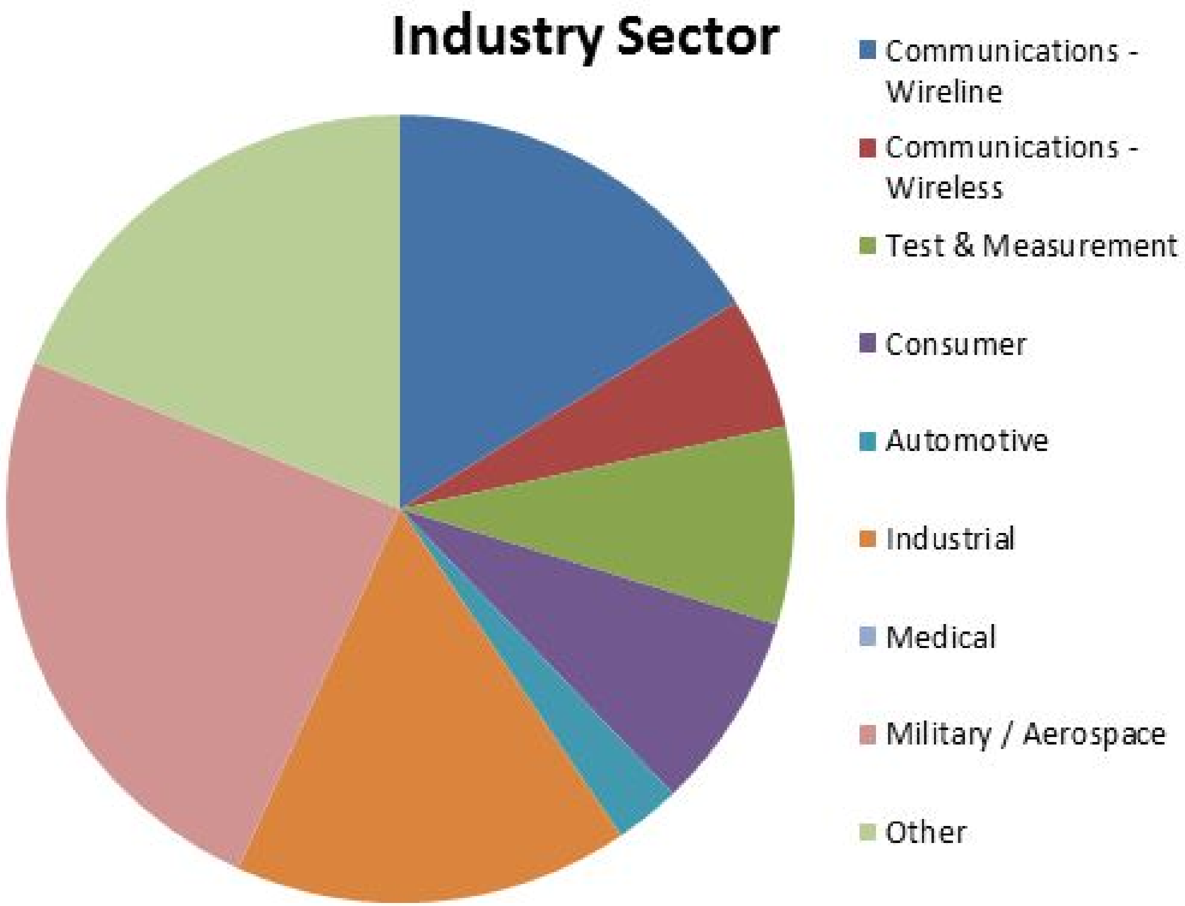


FPGA Industry



There are two major players in the FPGA market

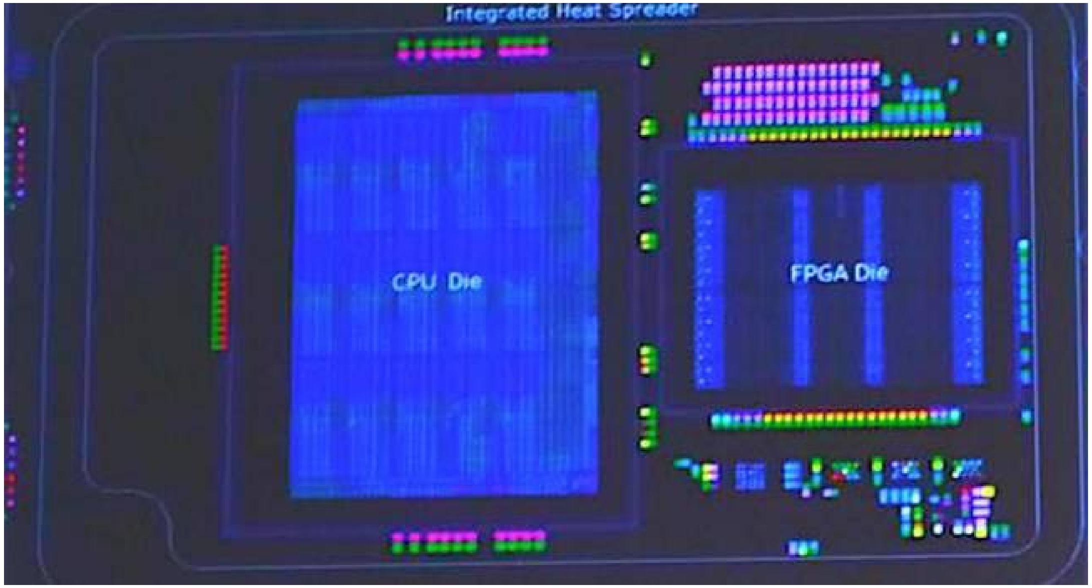
Where are FPGAs mostly used



So whats new?

December 2015 Intel buys Altera

Incoming Xeon processors, with FPGA coprocessors.



Spatial Computing Paradigms

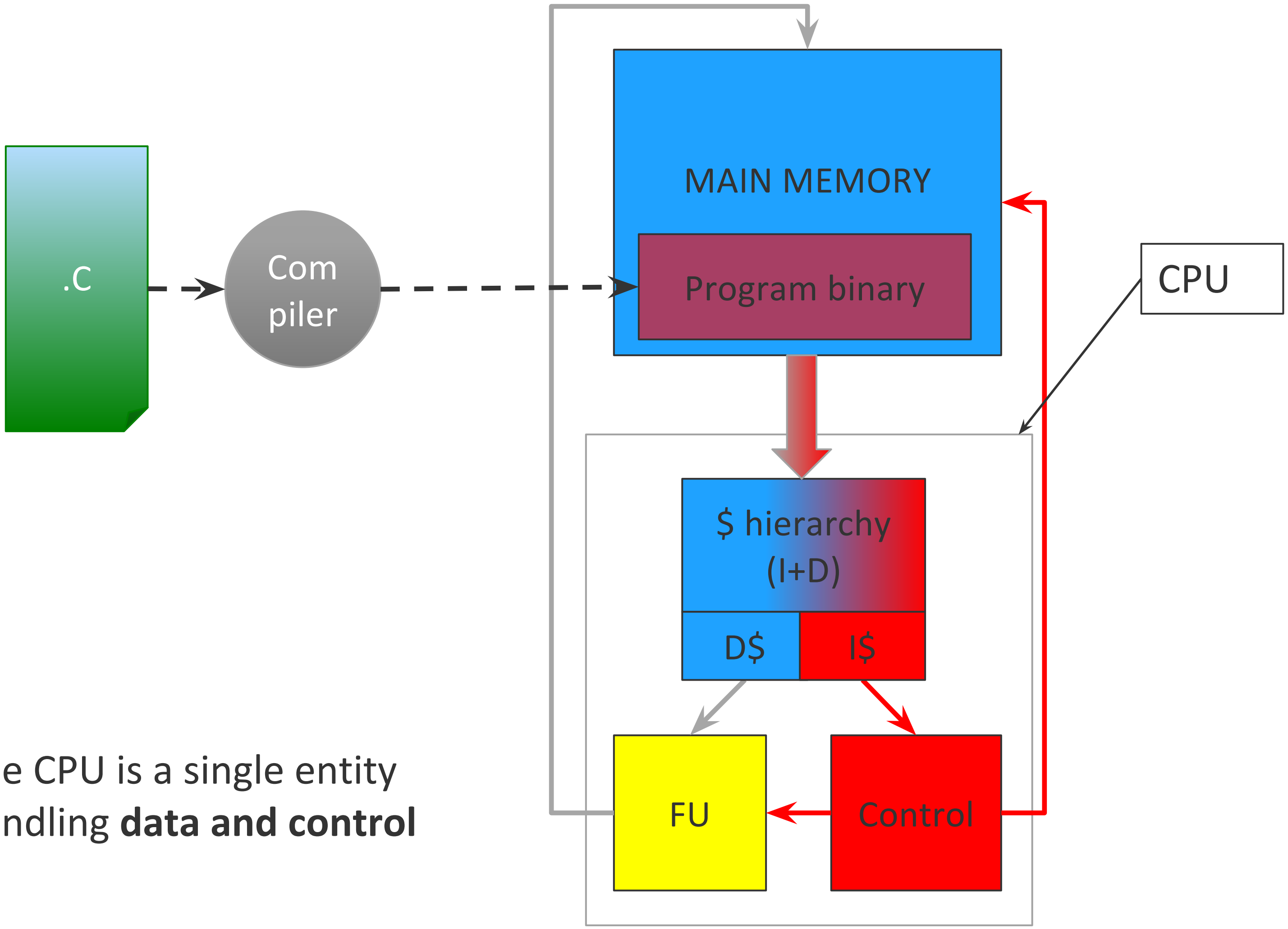
Computing with an FPGA requires a different mindset to ordinary software programming.

- We construct a deep pipeline (assembly line) on the substrate of the chip.
- Parallelism comes from arithmetic units each doing a small part of the work on some piece of data, then passing it on.
- The available space on the chip is finite, the pipeline must fit!
- We call this type of computing Dataflow (the data flows through the pipeline).

Vs SIMD

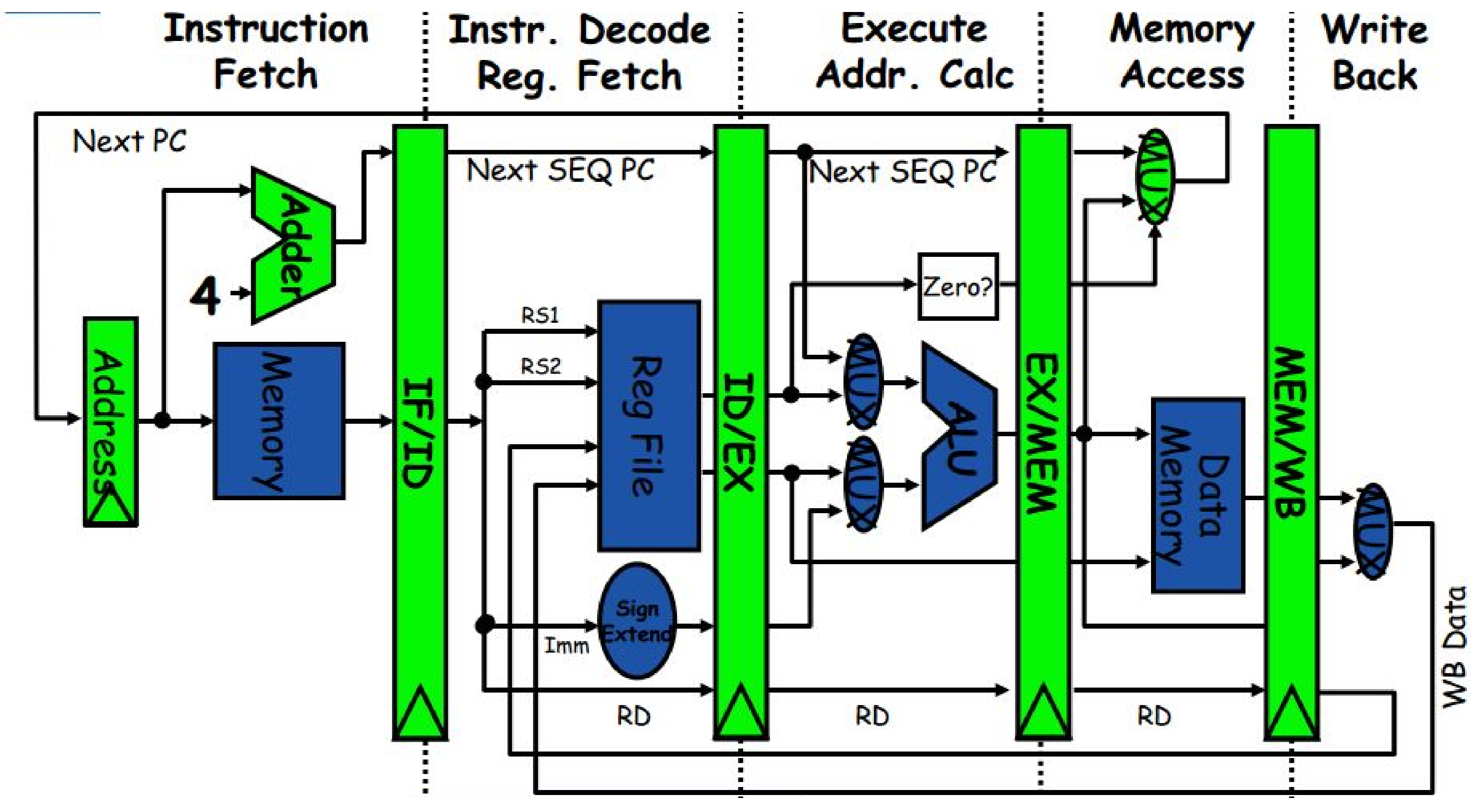
- Most often in Scientific Computing parallelism comes in the form of Single Instruction Multiple Dispatch.
- The same instruction is applied to many pieces of data (probably in an array), each thread gets one piece of data.
- Once this instruction has completed on all pieces of data a new instruction may be issued.
- In dataflow computing we may have Multiple Instruction Single Dispatch.
- A stream of data is passed through Multiple Instructions.

Control-flow Machine



The CPU is a single entity handling **data and control**

Simple CPU Pipeline



IR <= mem[PC];
PC <= PC + 4

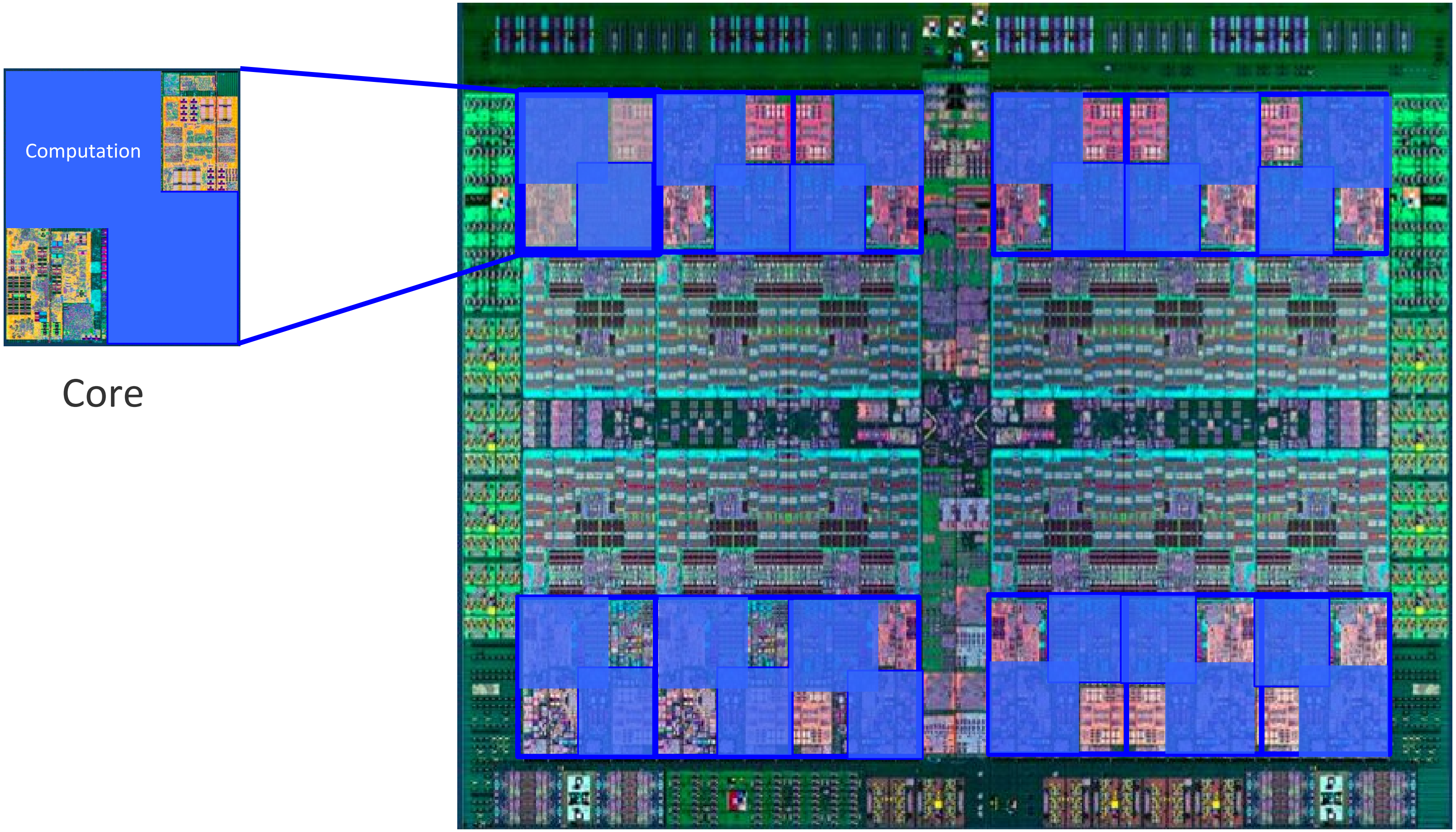
A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

rslt <= A op_{IRop} B

WB <= rslt

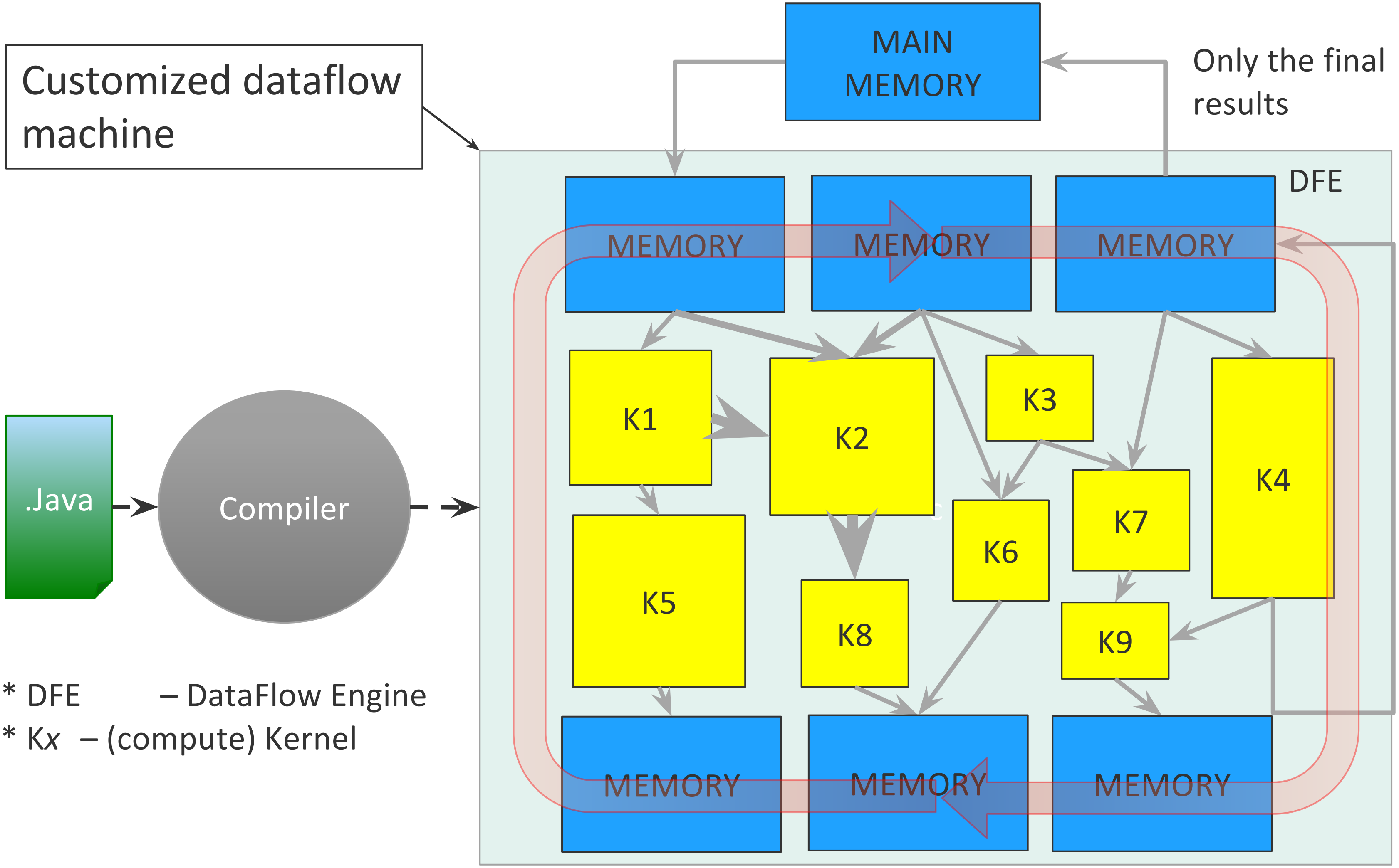
Reg[IR_{rd}] <= WB

Control-flow Computing example: IBM POWER 8, 12 cores @ 4 GHz



22nm SOI, eDRAM, 15 ML 650mm2,12 cores (SMT8)

Spatial Computing Machine



* DFE – DataFlow Engine
* K*x* – (compute) Kernel

Control Flow versus Data Flow

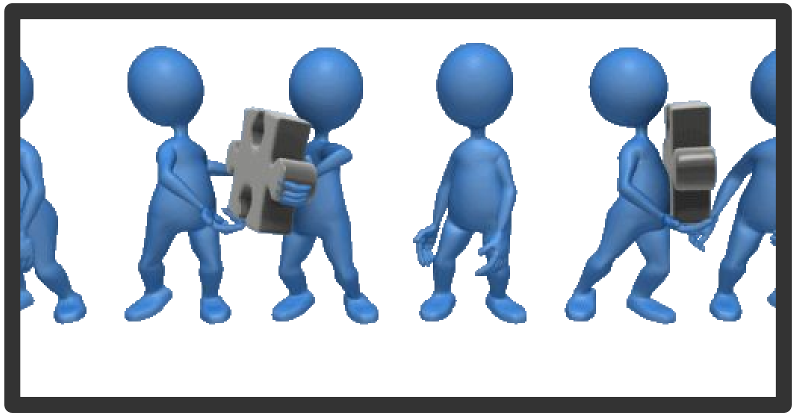
- Control Flow:
  - Instructions “move”
  - Data may move along with instructions (secondary issue)
  - Order of computation is the key

- Data Flow:
  - Data moves through a set of “instructions” in 2D(ish) space
  - Data moves will trigger control
  - Data availability, transformations and operation latencies are the key

Control Flow versus Data Flow

CPUs

FPGAs

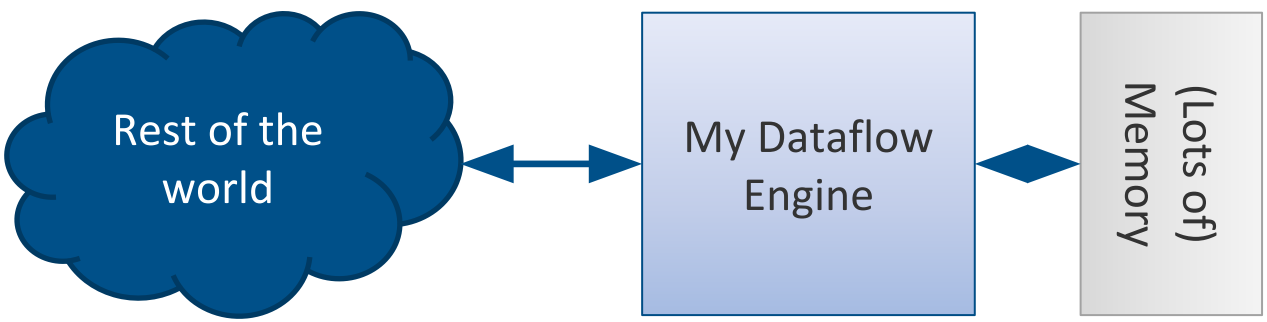


Data Flow specific properties

- No needed for:
  - shared memory
  - program counter
  - control sequencer
  - branch prediction
- Special mechanisms are required to:
  - data availability detection
  - orchestration of data tokens and “instructions”
  - chaining of asynchronous “instruction” execution

Dataflow Computing

- A custom chip for a specific application
- No instructions → no instruction decode logic
- No branches → no branch prediction
- Explicit parallelism → No out-of-order scheduling
- Data streamed onto-chip → No multi-level caches

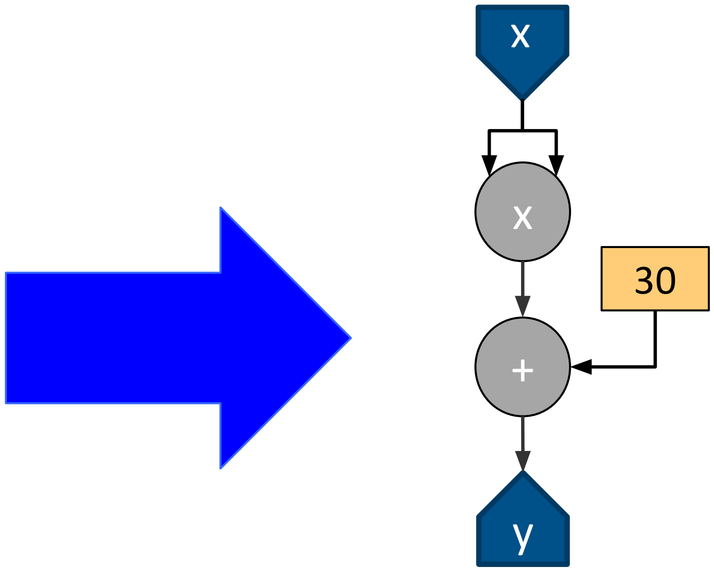


Converting Simple Expression

$$y_i = x_i \times x_i + 30$$

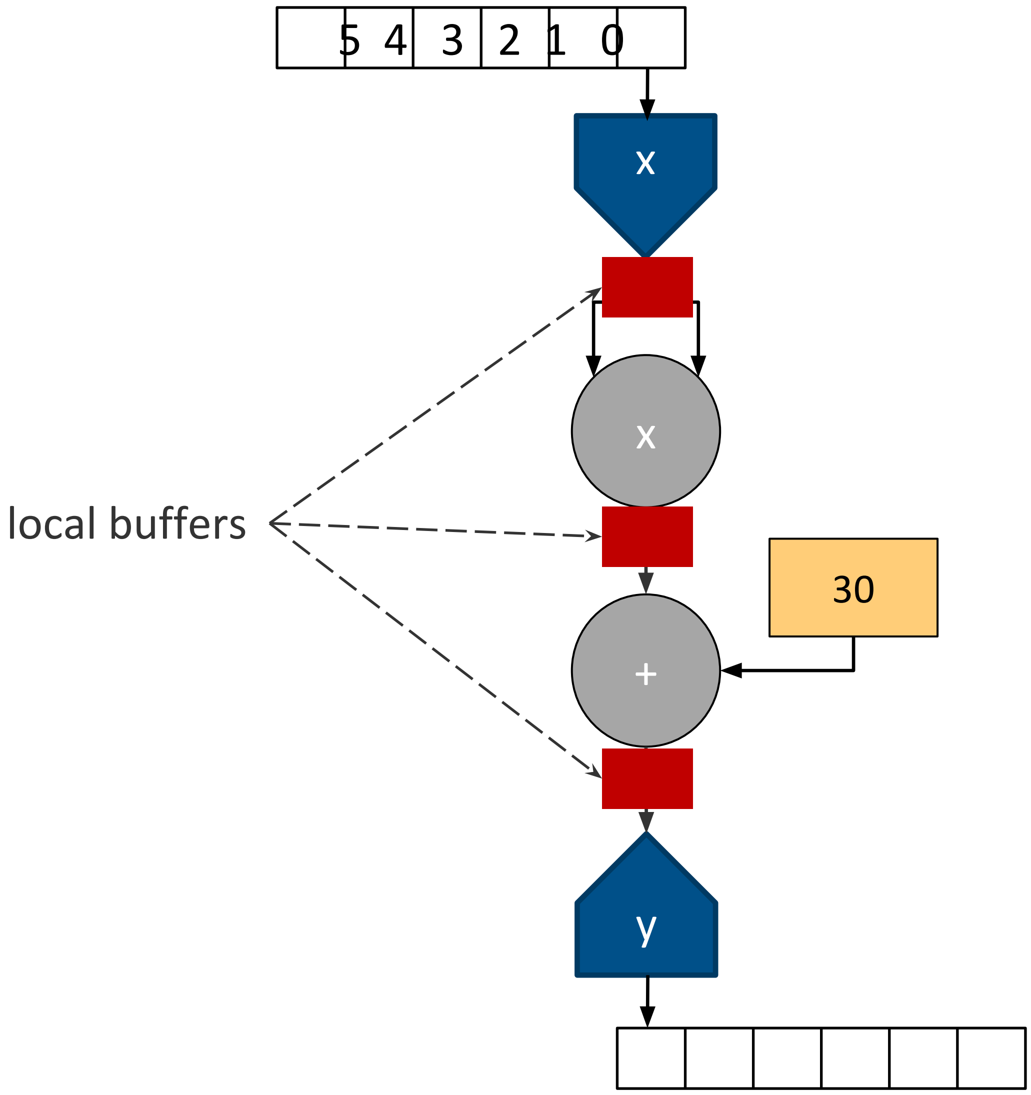
```
for (int i =0; i < DATA_SIZE; i++)
   y[i]= x[i] * x[i] + 30;
```

Input stream of integer elements ‘x’

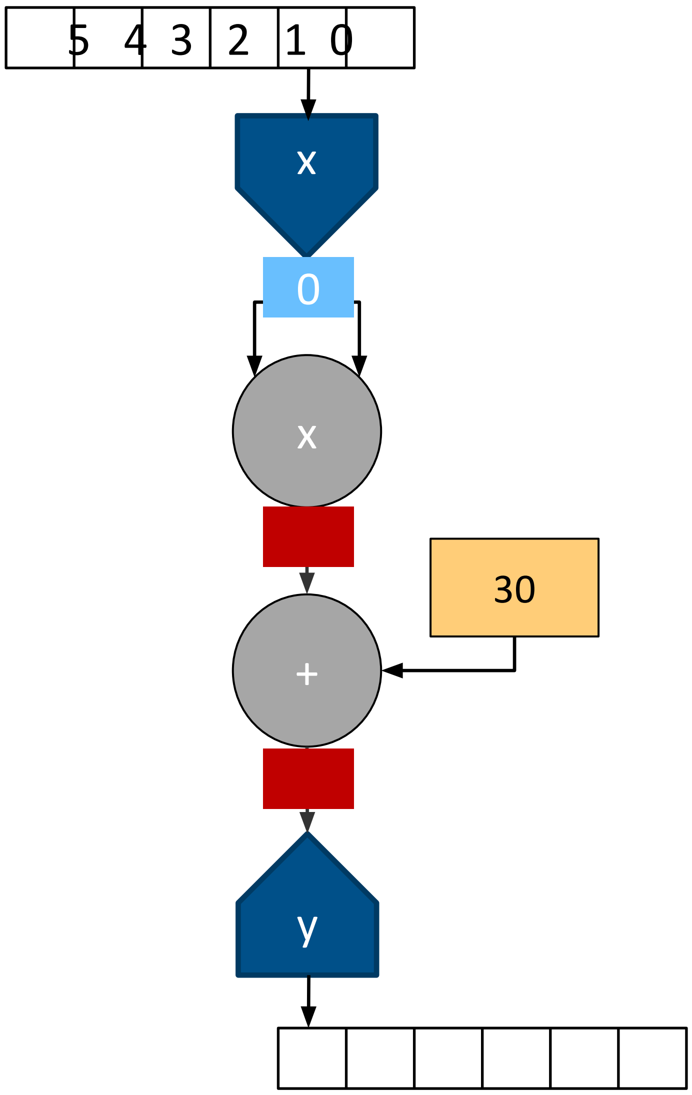


Output stream of integer elements ‘y’

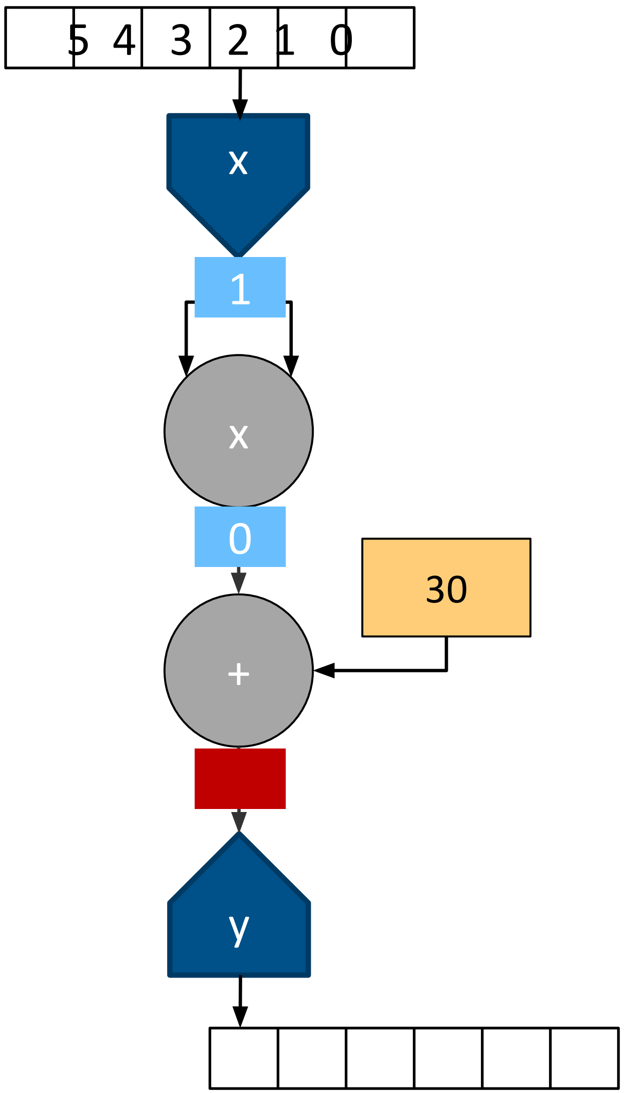
Flowing elements



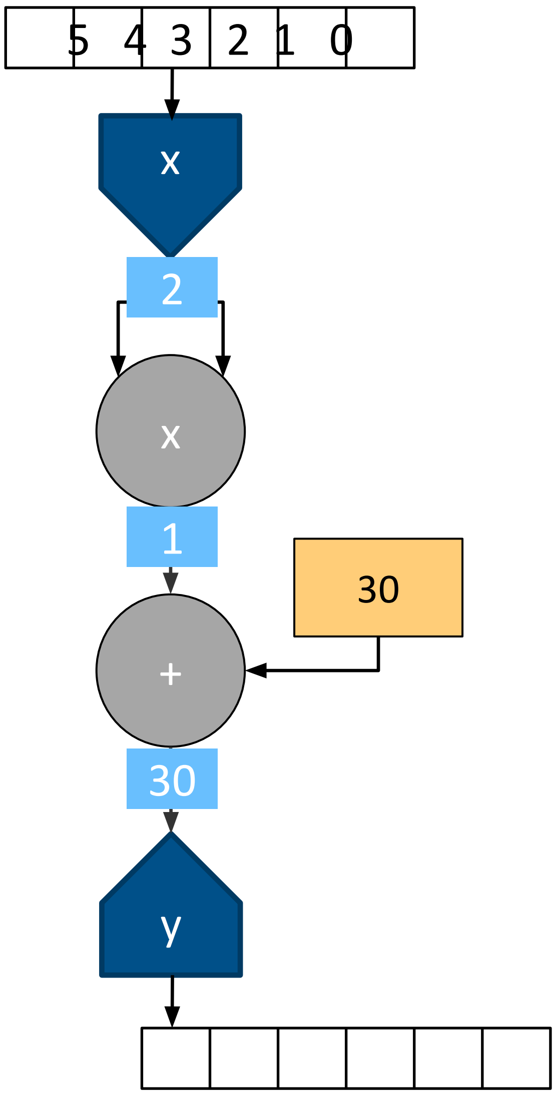
Flowing elements



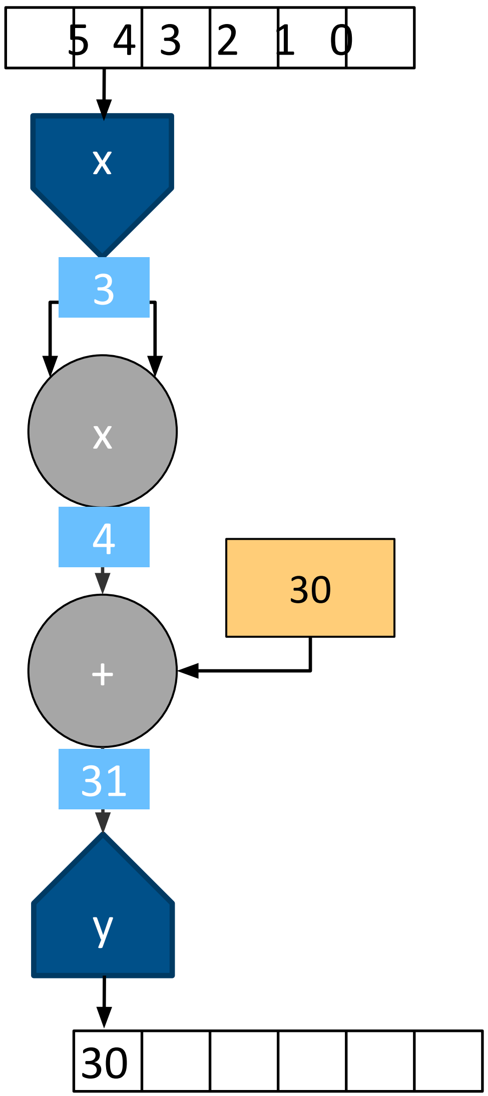
Flowing elements



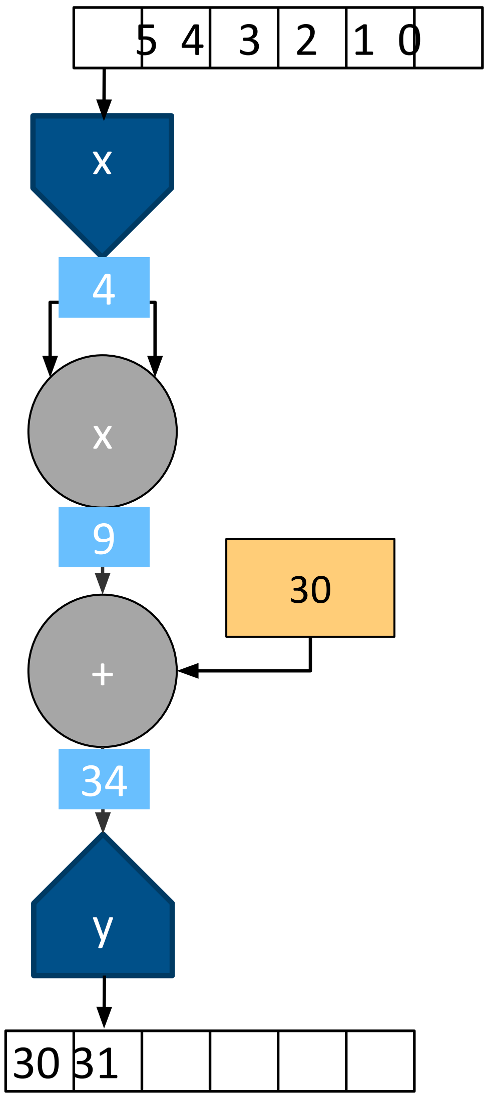
Flowing elements



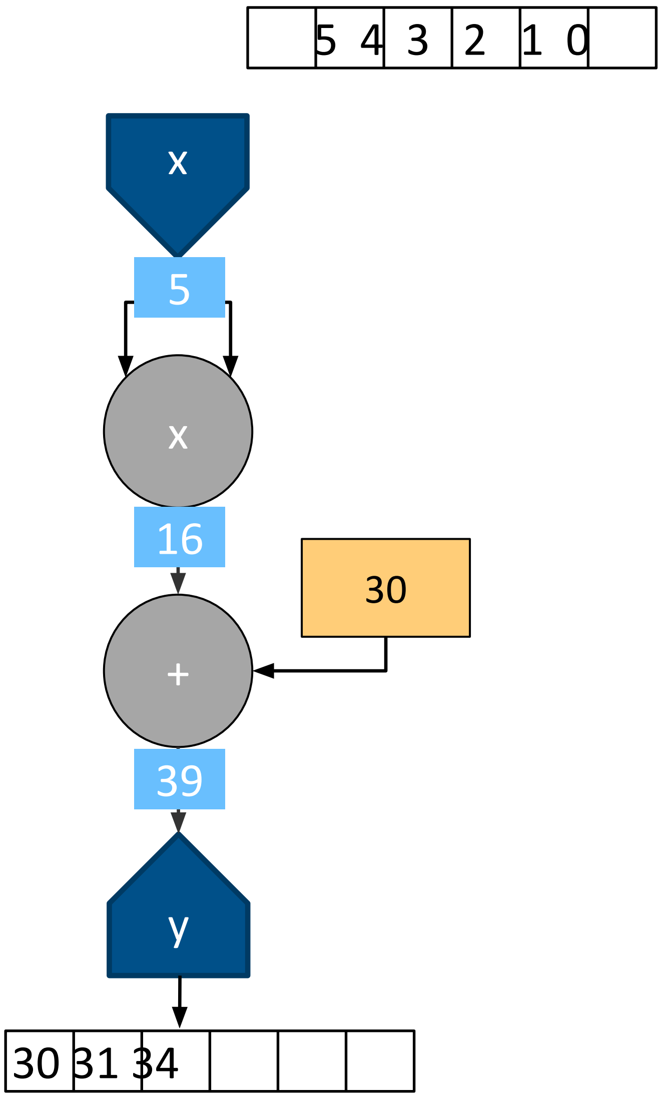
Flowing elements



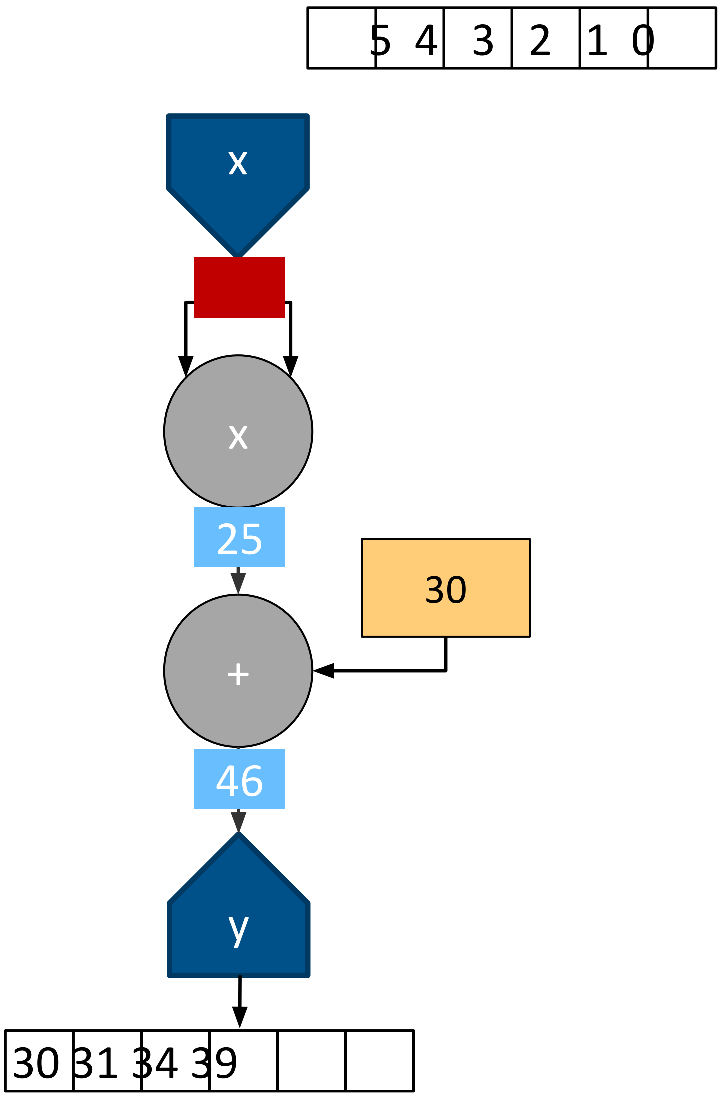
Flowing elements



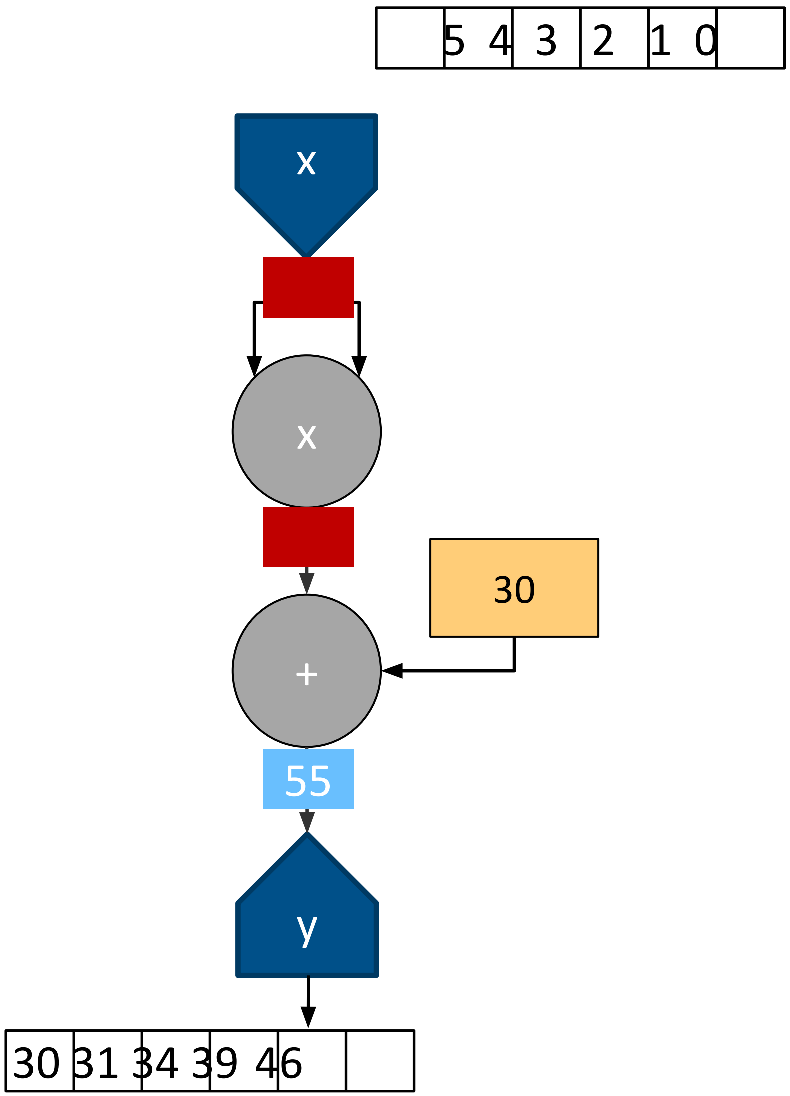
Flowing elements



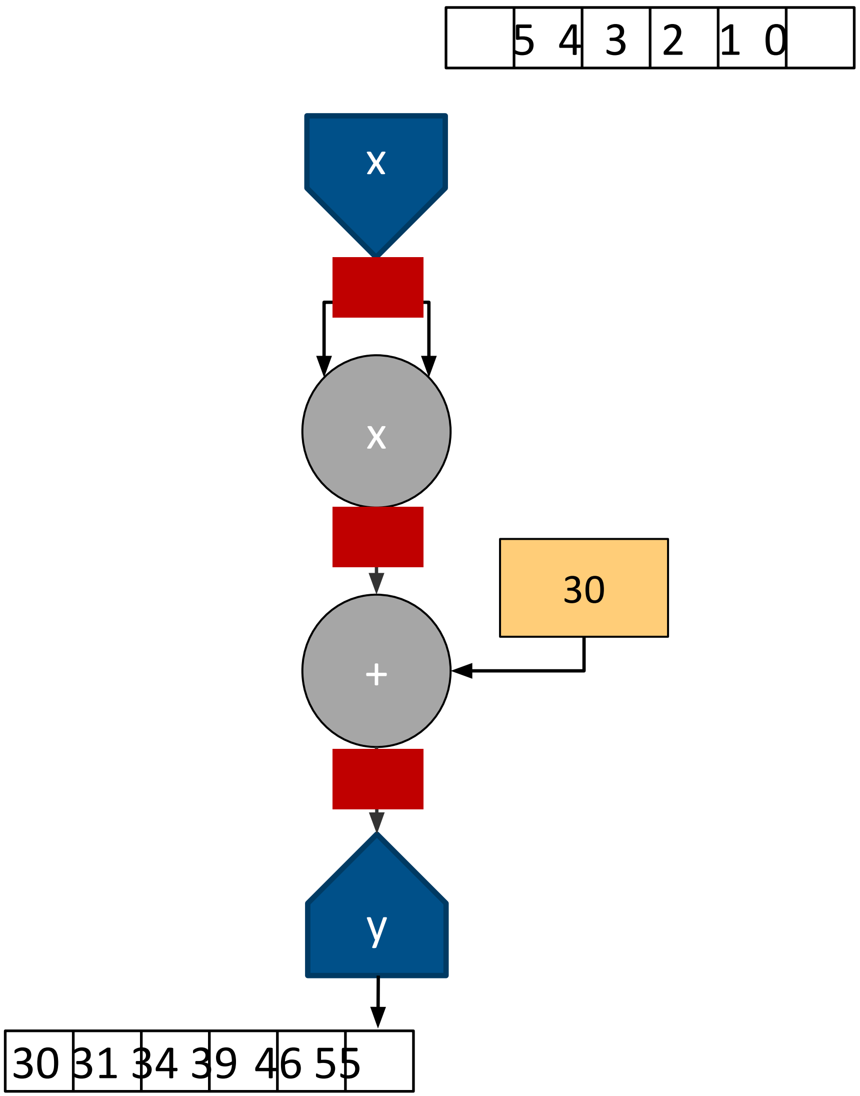
Flowing elements



Flowing elements

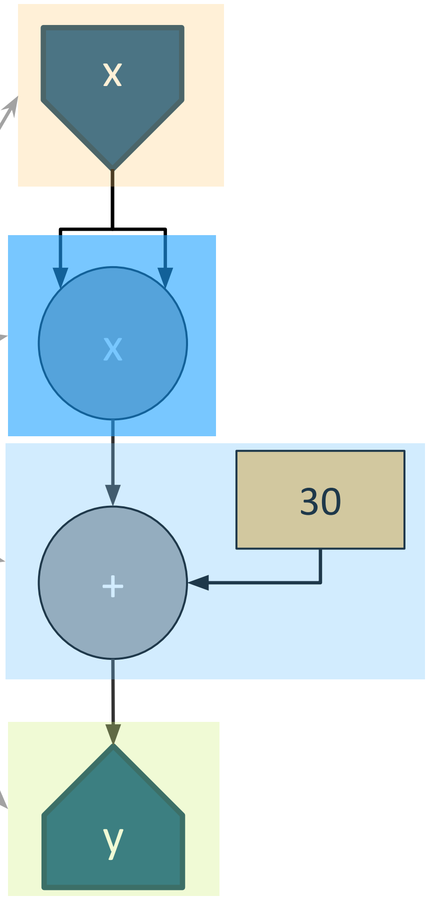


Flowing elements



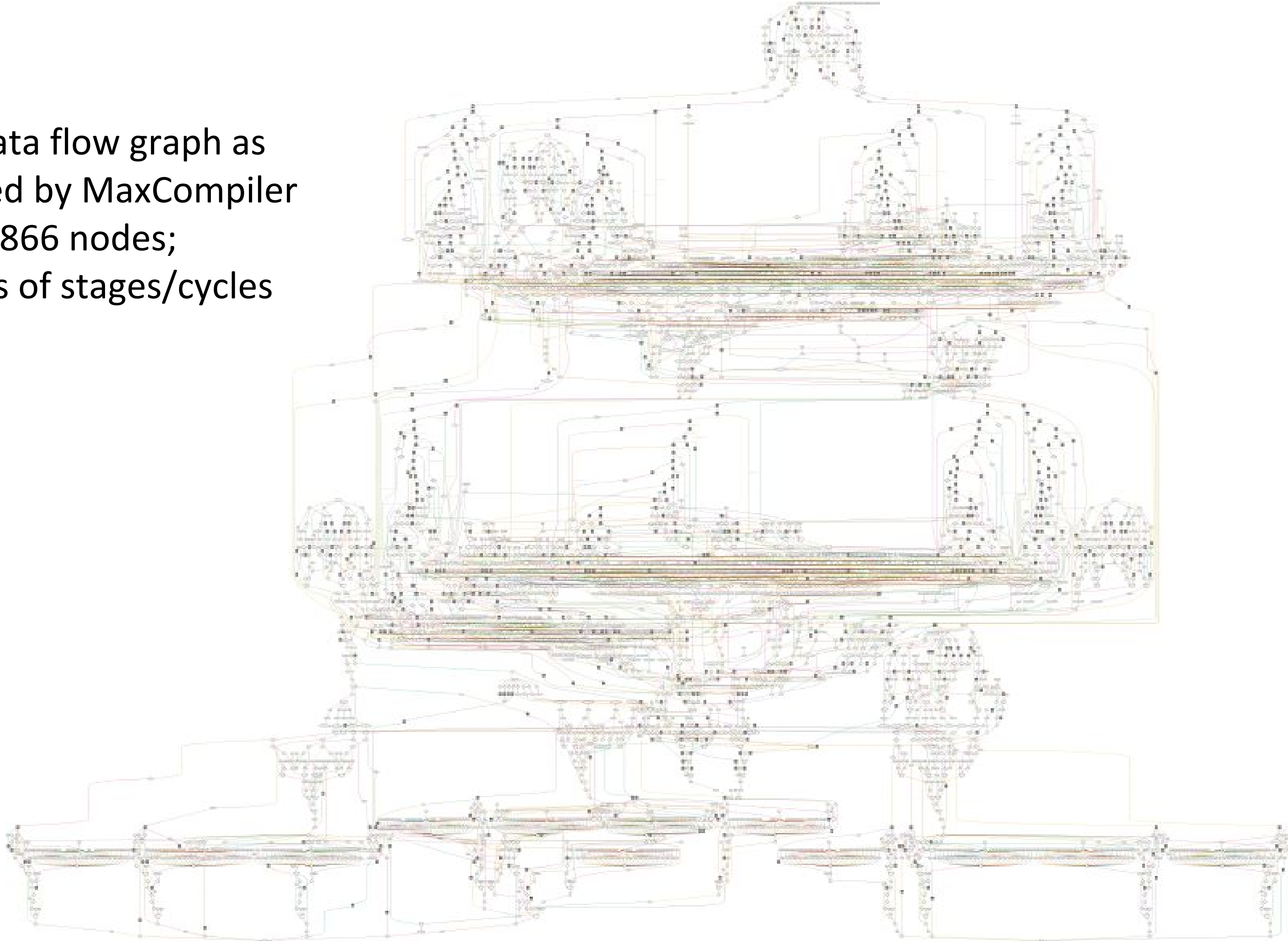
The Full Kernel

```
public class MyKernel extends Kernel {

    public MyKernel (KernelParameters parameters)
    {
        super(parameters);

        DFEVar x = io.input("x", dfeInt(32));

        DFEVar result = x * x + 30;

        io.output("y", result, dfeInt(32));
    }
}
```



Enabling large scale dataflow designs

Real data flow graph as generated by MaxCompiler
4866 nodes;
10,000s of stages/cycles



MAXELER Technologies

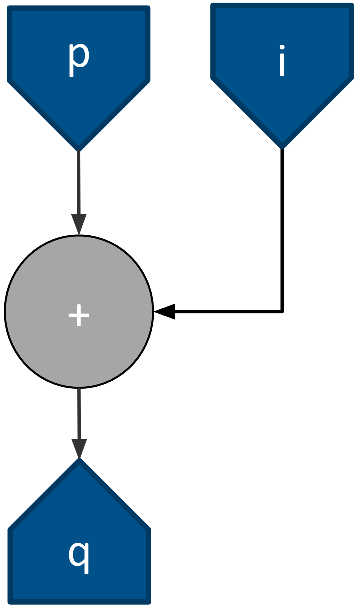
Generating data on chip

- How can we implement this?

```
for (int i = 0; i < N; i++) {
    q[i] = p[i] + i;
}
```

How about this?

```
DFEVar p = io.input("p", dfeInt(32));
DFEVar i = io.input("i", dfeInt(32));

DFEVar q = p + i;

io.output("q", q, dfeInt(32));
```

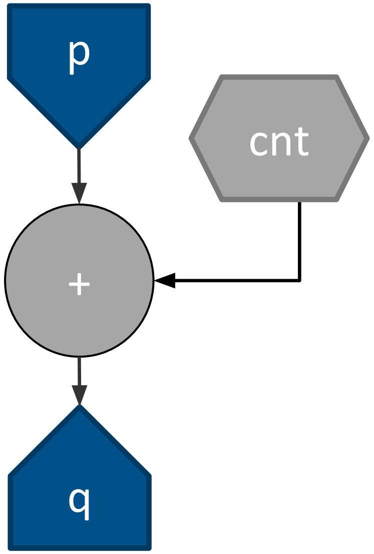


Yes…. But, now we need to create an array *i* in software and send it to the DFE as well

Generating data on chip

- There is very little 'information' in the *i* stream.
  - Could compute it directly on the DFE itself

```
DFEVar p = io.input("p", dfeInt(32));
DFEVar i = control.count.simpleCounter(32, N);

DFEVar q = p + i;

io.output("q", q, dfeInt(32));
```



Half as many inputs
Less data transfer

- Counters can be used to generate sequences of numbers
- Complex counters can have strides, wrap points, triggers:
  - E.g. *if (y==10) y=0; else if (en==1) y=y+2;*

Stream Offsets

- So far, we've only performed operations on each individual point of a stream
  - The stream size doesn't actually matter (functionally)!
  - At each point computation is independent
- Real world computations often need to access values from more than one position in a stream
  - For example, a 3-pt moving average filter:

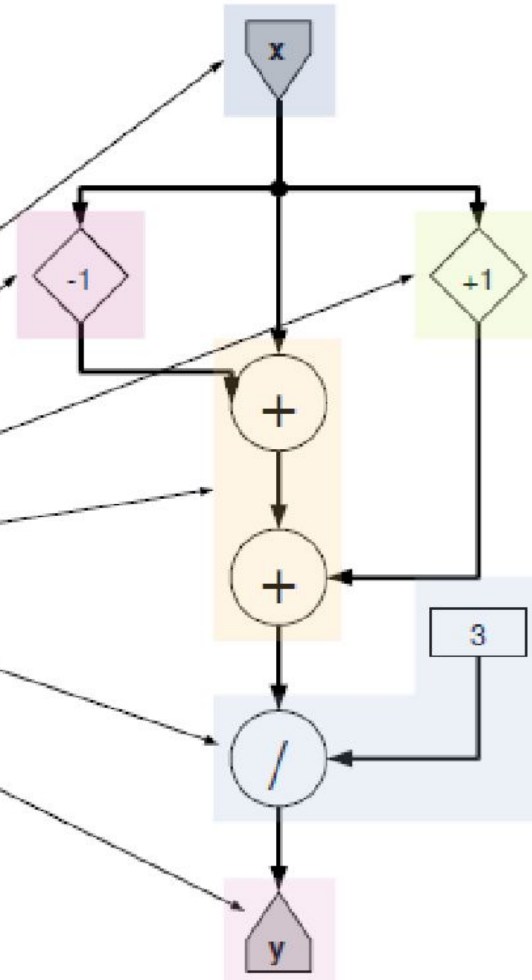
$$y_i = (x_{i-1} + x_i + x_{i+1}) / 3$$

Stream Offsets

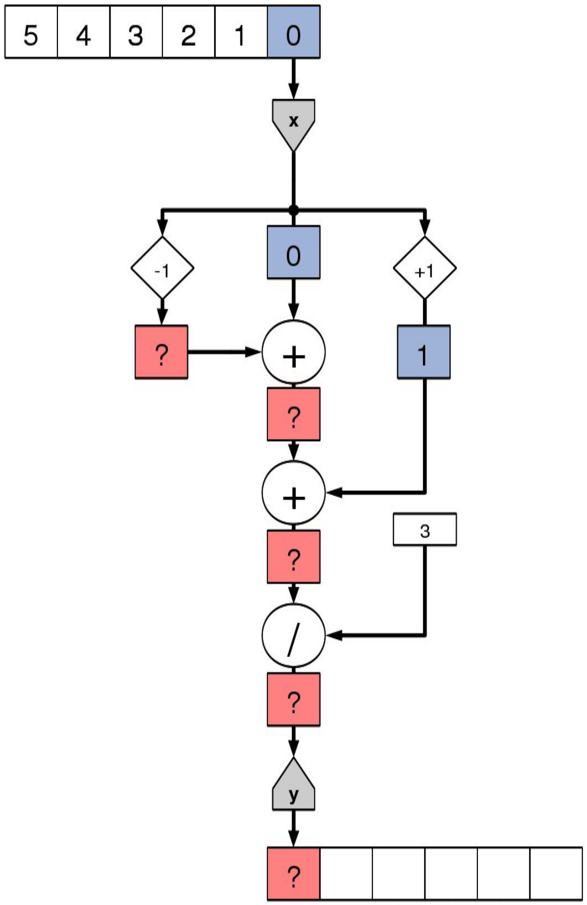
- *Stream offsets* allow us to compute on values in a stream other than the current value.
- Offsets are relative to the *current position* in a stream; *not* the start of the stream
- Stream data will be buffered on-chip in order to be available when needed → uses fast memory (FMEM)
  - Maximum supported offset size depends on the amount of on-chip SRAM available. Typically 10s of thousands of points.

Moving Average in MaxCompiler

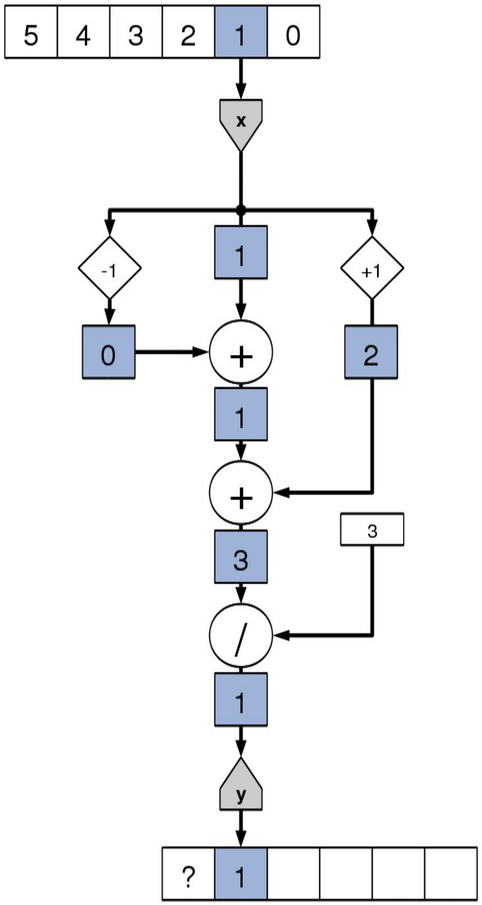
```
class MovingAverageSimpleKernel extends Kernel {

    MovingAverageSimpleKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar x = io.input("x", dfeFloat(8, 24));

        DFEVar prev = stream.offset(x, −1);
        DFEVar next = stream.offset(x, 1);
        DFEVar sum = prev + x + next;
        DFEVar result = sum / 3;

        io.output("y", result, dfeFloat(8, 24));
    }
}
```



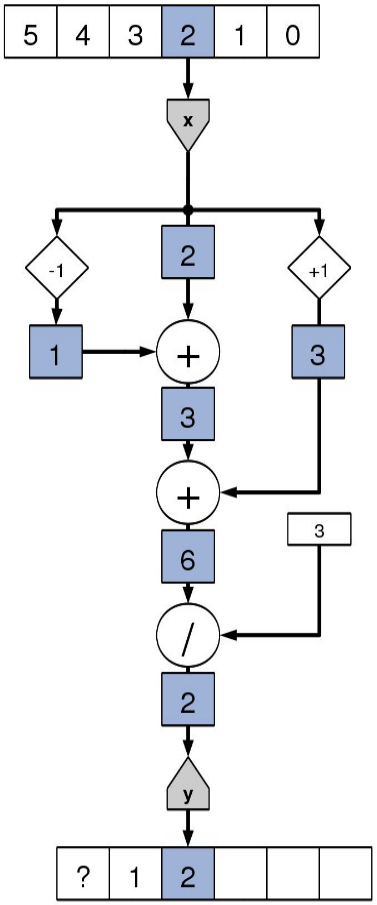
Kernel Execution



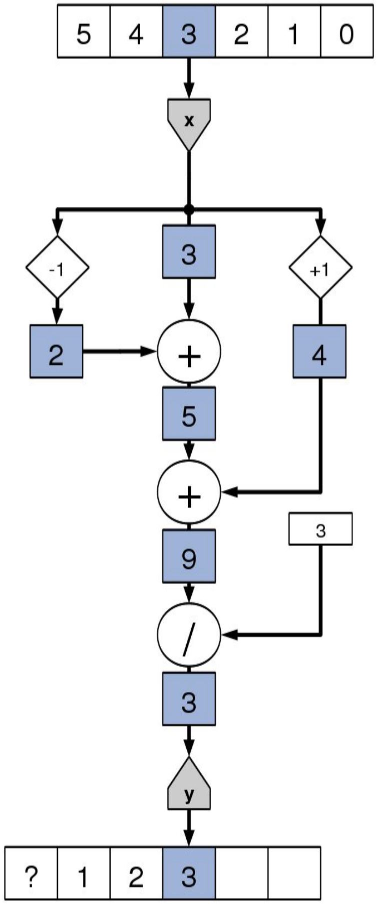
Kernel Execution



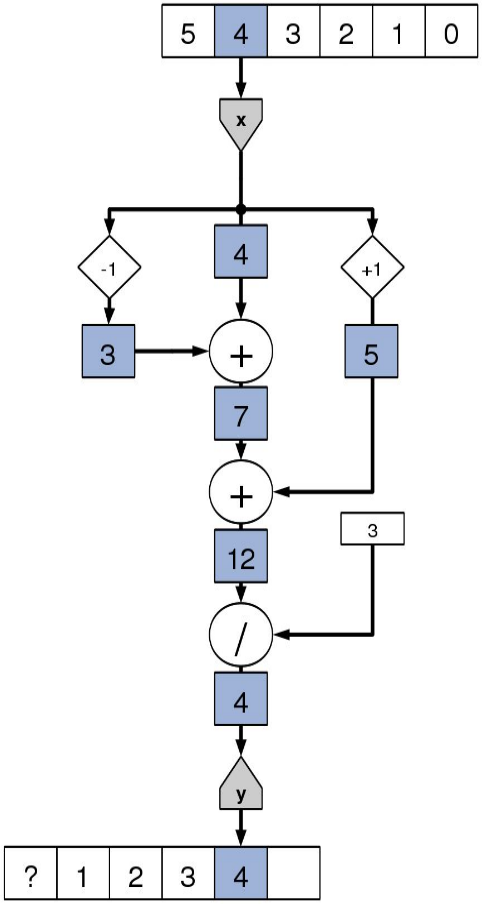
Kernel Execution



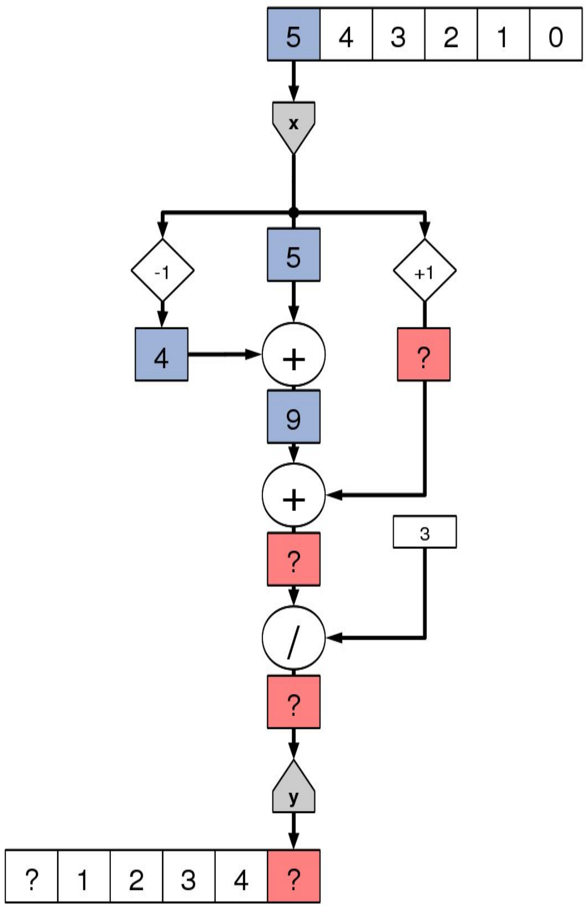
Kernel Execution



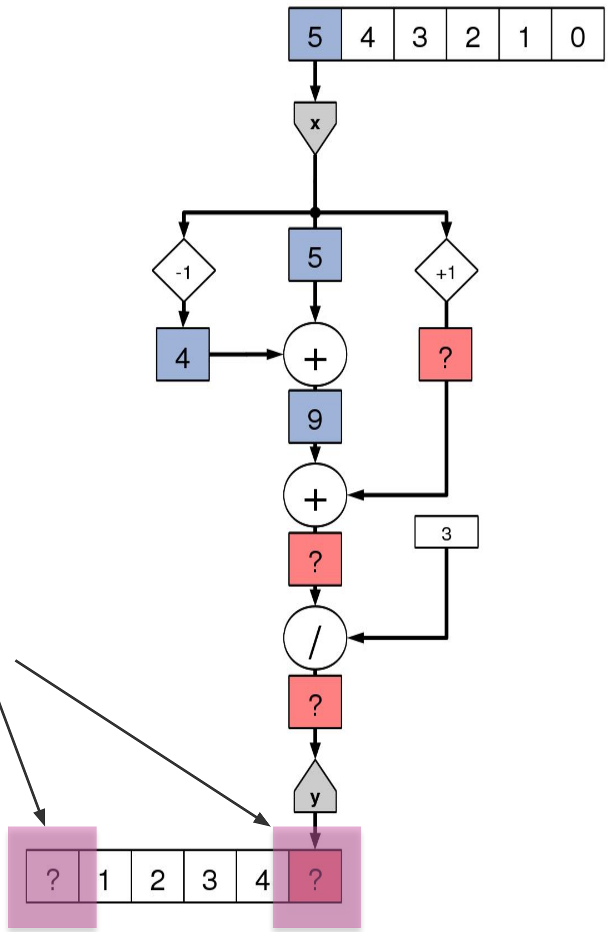
Kernel Execution



Kernel Execution



Boundary Cases



What about the boundary cases?

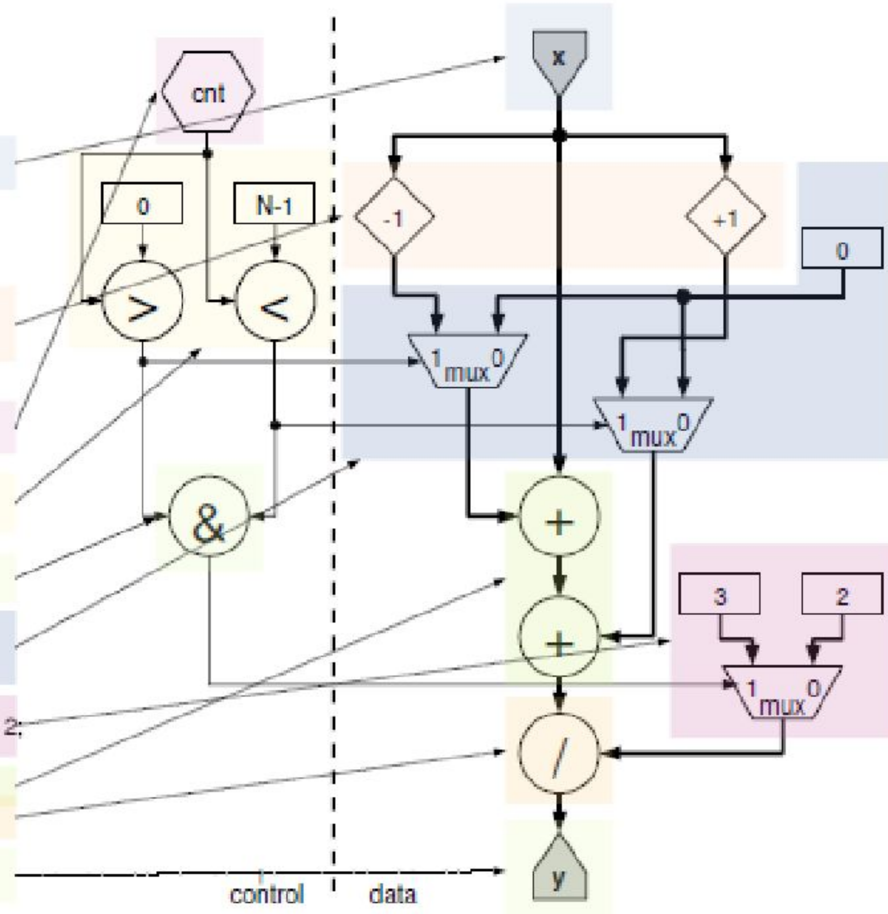
More Complex Moving Average

- To handle the boundary cases, we must explicitly code special cases at each boundary

$$y_i = \begin{cases} (x_i + x_{i+1})/2 & \text{if } i = 0 \\ (x_{i-1} + x_i)/2 & \text{if } i = N-1 \\ (x_{i-1} + x_i + x_{i+1})/3 & \text{otherwise} \end{cases}$$

Kernel Handling Boundary Cases

```
class MovingAverageKernel extends Kernel {

    MovingAverageKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar x = io.input("x", dfeFloat(8, 24));

        DFEVar size = io.scalarInput("size", dfeUInt(32));

        // Data
        DFEVar prevOriginal = stream.offset(x, -1);
        DFEVar nextOriginal = stream.offset(x, 1);

        // Control
        DFEVar count = control.count.simpleCounter(32, size);

        DFEVar aboveLowerBound = count > 0;
        DFEVar belowUpperBound = count < size - 1;

        DFEVar withinBounds = aboveLowerBound & belowUpperBound;

        DFEVar prev = aboveLowerBound ? prevOriginal : 0;
        DFEVar next = belowUpperBound ? nextOriginal : 0;

        DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;

        DFEVar sum = prev + x + next;
        DFEVar result = sum / divisor;

        io.output("y", result, dfeFloat(8, 24));
    }
}
```



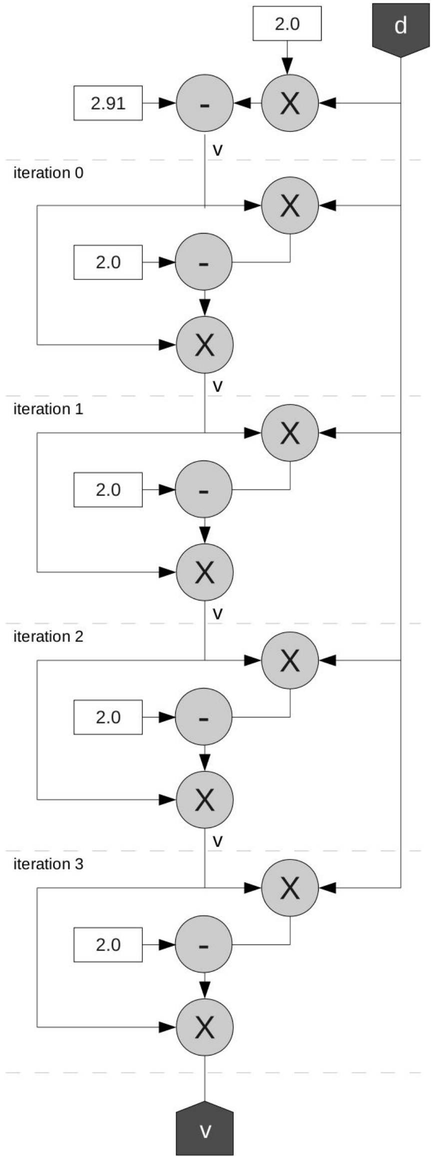
Starting on Scientific Computing

- Often in scientific computing, compute may be structured as nested loops.
- On FPGA the length of these for loops becomes critical.
- The reason for this is that the space on the chip is limited, at some point there will be a cutoff where the loop is too large to be unrolled.
- Now follows some discussion on the types of cases which may occur.

Loop Unrolling in space with Dependence

```
for (i = 0; ; i += 1)  {
   float d = input[i];
   float v = 2.91 - 2.0*d;
   for (iter=0; iter < 4; iter += 1)
        v = v * (2.0 - d * v);
   output[i] = v;
}
```

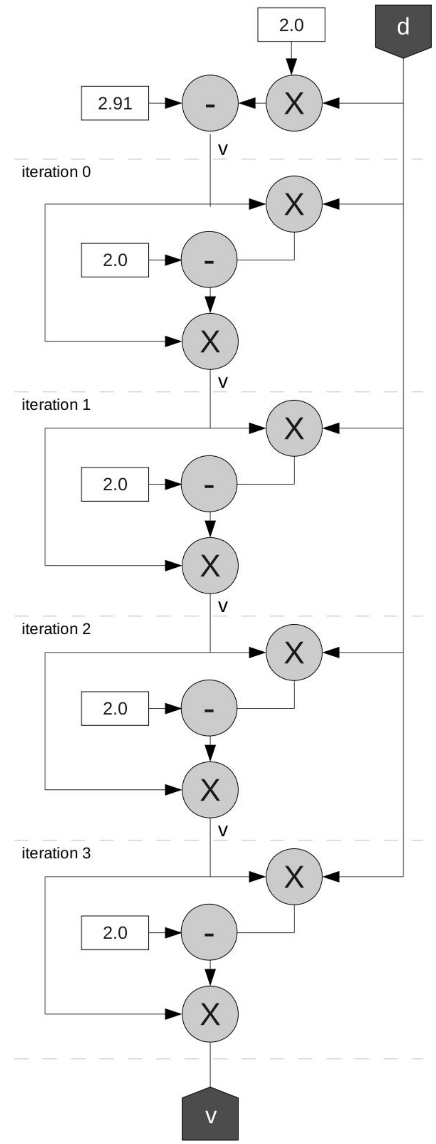
```
DFEVar d = io.input("d", dfeFloat(8, 24));
DFEVar TWO= constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) − TWO*d;

for ( int iteration = 0; iteration < 4; iteration += 1) {
        v = v*(TWO− d*v);
}
io.output("output" , v, dfeFloat(8, 24));
```



Loop Unrolling with Dependence

```
float d = input;
float v = 2.91 - 2.0*d;
for (iter=0; iter < 4; iter += 1)
  v = v * (2.0 - d * v);
output = v;
```

```
DFEVar d = io.input("d", dfeFloat(8, 24));
DFEVar TWO= constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) − TWO*d;

for ( int iteration = 0; iteration < 4; iteration += 1) {
        v = v*TWO− d*v;
}
io.output("output" , v, dfeFloat(8, 24));
```



- **The software loop has a cyclic dependence (v)**
- **But the unrolled datapath is acyclic**

Variable Length Loop

```
int d = input;
int shift = 0;
while (d != 0 && ((d & 0x3FF) != 0x291)) {
 shift = shift + 1;
 d = d >> 1;
}
output = shift;
```

- What do we do with a while loop (or a loop with a “break”)?

```
// converted to fixed length
int d = input;
int shift = 0;
bool finished = false;
for (int i = 0; i < 22; ++i) {
  bool condition = (d != 0 && ((d & 0x3FF) != 0x291));
  finished = condition ? true : finished; // loop-carried
  shift = finished ? shift : shift + 1;   // dependencies
  d = d >> 1;
}
output = shift;
```

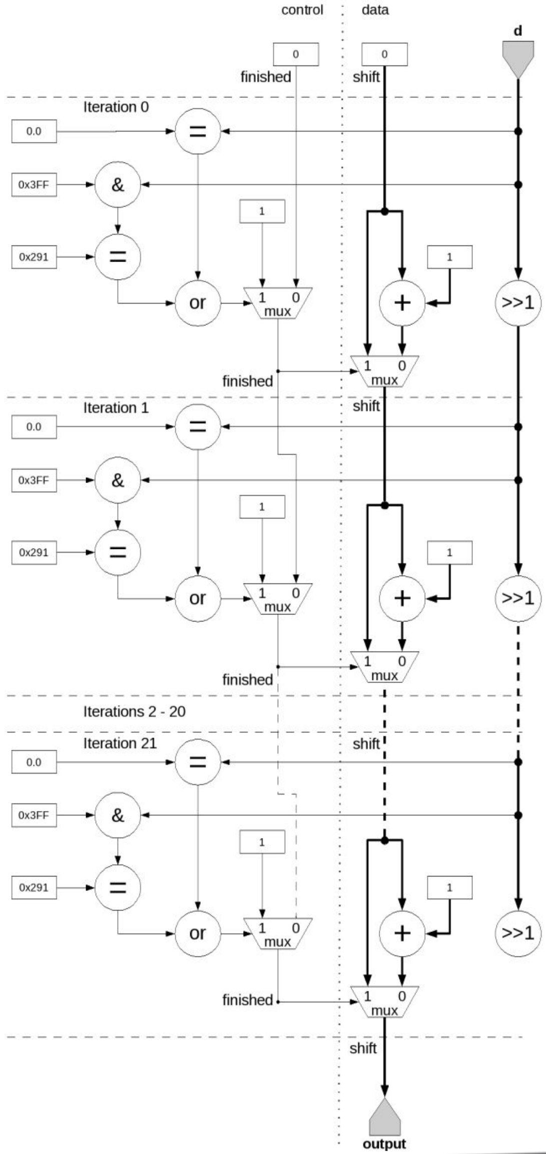
- Find maximum number of iterations
- *Predicate* execution of loop body
- Using a bool that is set to false when the while loop condition fails

i	Condition	Finished	Shift
1	f	f	1
2	f	f	2
3	f	f	3
4	f	f	4
5	**t**	**t**	5
6	f	**t**	5
7	f	**t**	5
8	f	**t**	5
9	f	**t**	5

Variable Length Loop – in hardware

```
int d = input;
int shift = 0;
bool finished = false;
for (int i = 0; i < 22; ++i) {
  bool condition=(d!=0&&((d&0x3FF)!=0x291));
  finished = condition ? true : finished;
  shift = finished ? shift : shift + 1;
  d = d >> 1;
}
int output = shift;
```

```
DFEVar d = io.input(”d”, dfeUInt(32));
DFEVar shift = constant.var(dfeUInt(5), 0);
DFEVar finished = constant.var(dfeBool(), 0);
for ( int i = 0; i < 22; ++i) {  // unrolled
        DFEVar condition = d.neq(0)&((d&0x3FF).neq(0x291));
        finished = condition ? constant.var(1) : finished ;
        shift = finished ? shift : shift + constant.var(1);
        d = d >> 1;
}
io.output(”output” , shift , dfeUInt(5));
```



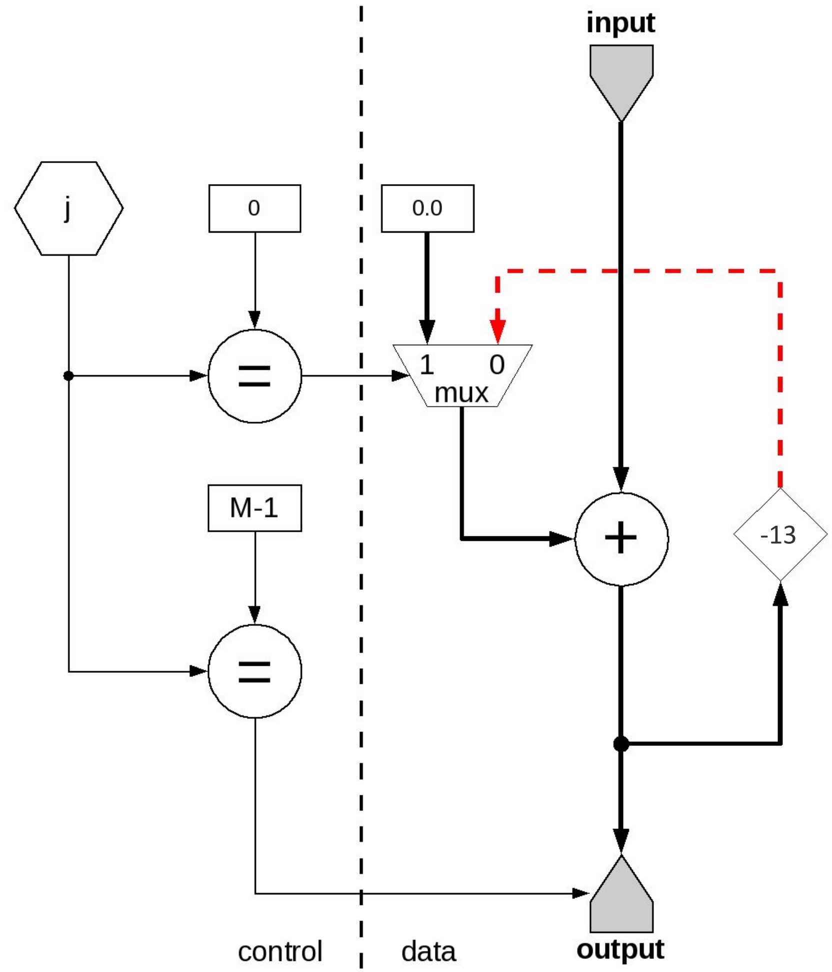
To Unroll or Not to Unroll

- Loop Unrolling
  - Gets rid of loop-carried dependency by creating a long pipeline
  - Requires O(N) space on the chip...what if it does not fit?
  - If we can’t unroll, we end up with a cycle in the dataflow graph
  - As we will see, we need to take care to make sure the cycle is compatible with the pipeline depth

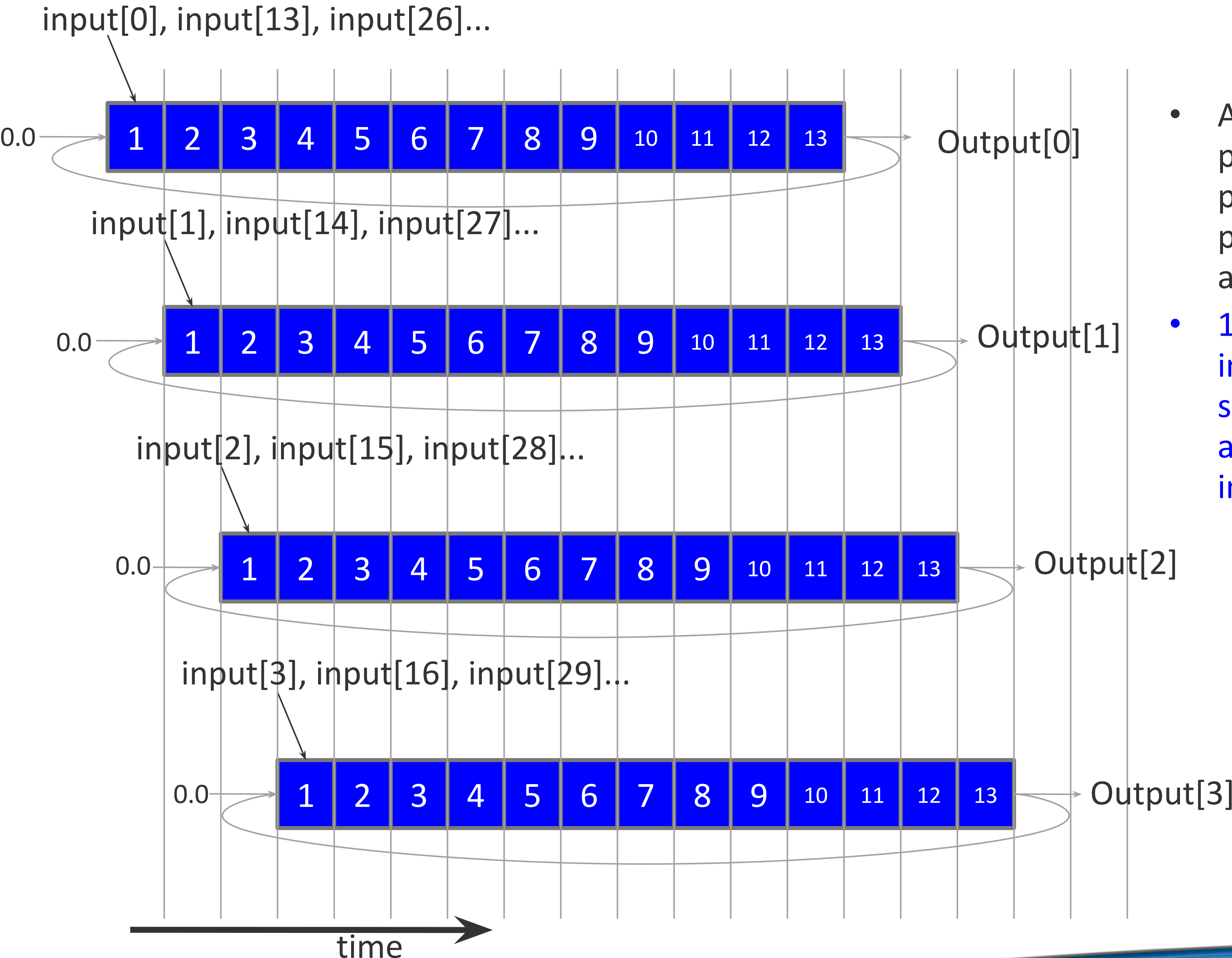
- Variable-length loop (with loop-carried dependency)
  - Can be fully unrolled, BUT need to know maximal number of iterations
  - Utilization depends on actual data...
  - What if max iterations is much larger than average? Or max is not known? Or max iterations don’t fit on the chip?

Unrolling in time - Acyclic pipeline

```
sum = 0.0;
for (int j=0; j<M; j += 1) {
 sum = sum + input[j];
}
output = sum;
```

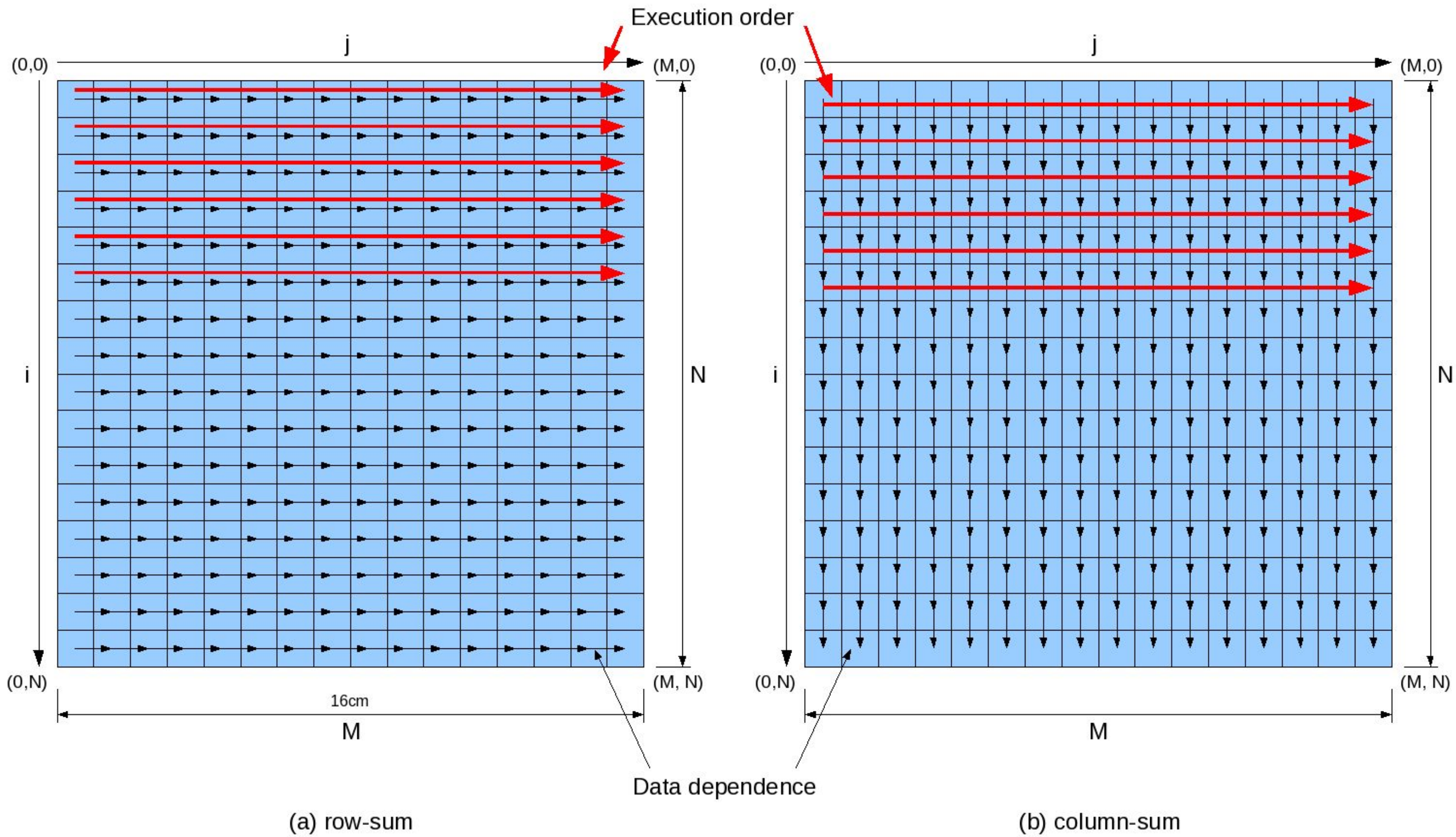


- Carrying dependency across cycles is quite different.
- A floating point adder takes 12 cycles, and a mux one.
- Hence the mux plus add takes 13 cycles, we can only receive an input every 13 cycles.
- This poor throughput is unacceptable.
- The answer is to do 13 partial sums.



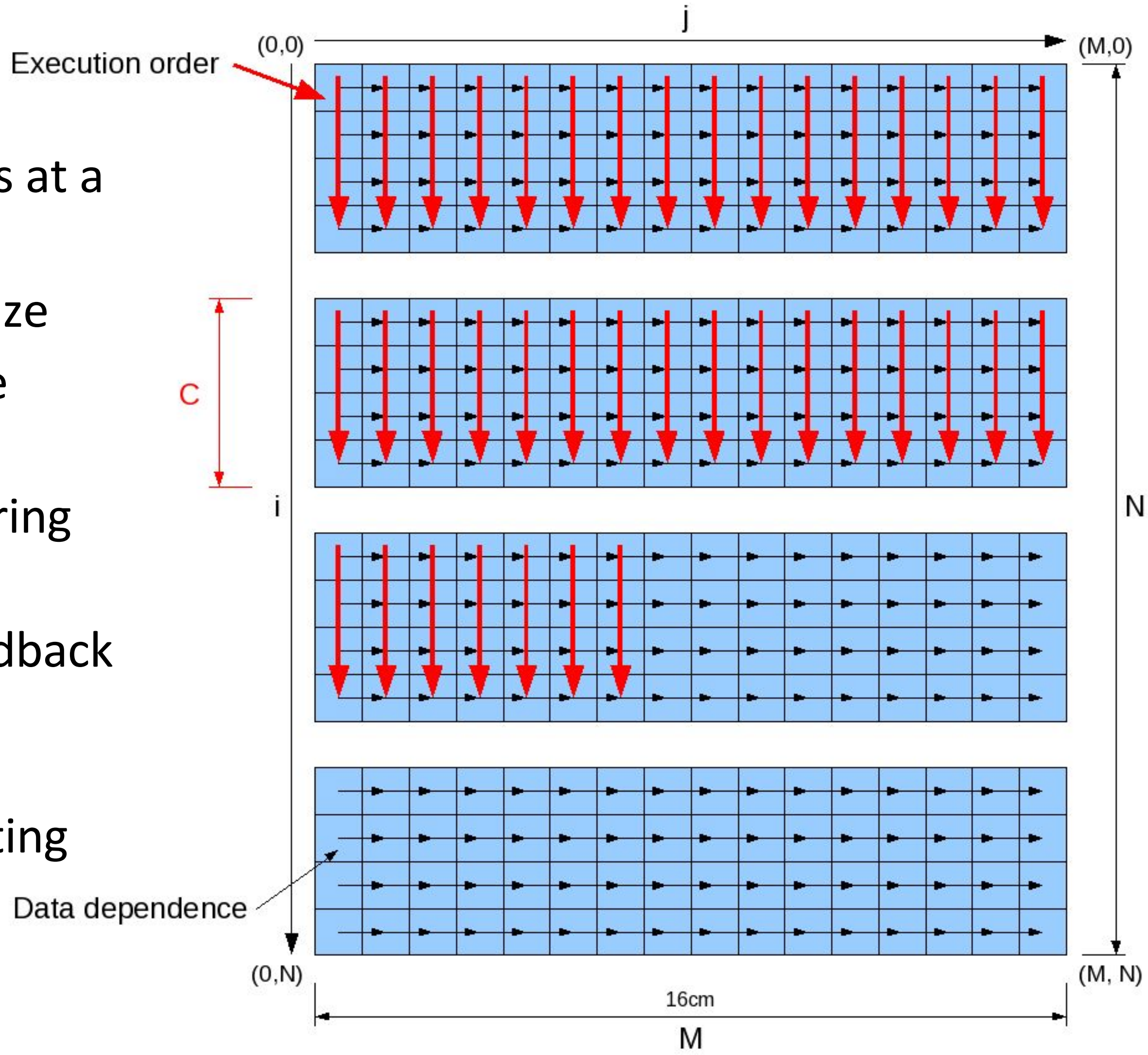
- After an initial pipeline fill phase, all 13 pipeline stages are occupied
- 13 independent summations are computed in parallel

Towards some Linear Algebra



- **Example**: Row-wise summation is serial due to chain of dependence
- Column-wise summation would be easy
- So we can keep the pipeline in a cyclic data datapath full by flipping the problem – ie by interchanging the loops

Multiple row sums simultaneously using one adder



- Idea: sum a block of rows at a time ("tiling")
- We can choose the tile size
- Just big enough to fill the pipeline
- so no unnecessary buffering is needed
- c is the length of the feedback loop, depending on the number format for the accumulator (12 for floating point).

Number Representation

- Microprocessors:
  - Integer: unsigned, one's complement, two's complement,
  - Floating Point: IEEE single-precision, double-precision

- Others:
  - Fixed point
  - Logarithmic number representation
  - Redundant number systems: use more bits, compute faster
    - Signed-digit representation
    - Residue number system (modulo arithmetic)
  - Decimal: decimal floating point, binary coded decimal

Fixed Point Numbers

- Generalisation of integers, with a ‘radix point’
- Digits to the right of the radix point represent negative powers of 2

Digit weights: (unsigned)

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}

I bits | *F* bits

- *F* = number of fractional bits
  - Bits to the right of the ‘radix point’
  - For integers, F = 0

Fixed Point Mathematics

- Think of each number as: $(V \times 2^{-F})$
- Addition and subtraction: $(V1 \times 2^{-F1}) + (V2 \times 2^{-F2})$
  - Align radix points and compute the same as for integers

	1	0	0	1	0	•	1	0	0	1	0	
+		1	0	1	1	•	0	1	0	0	1	0
=	1	1	1	0	1	•	1	1	0	1	1	0

- Multiplication: (V1 × 2) × (V2 × 2) = V1×V2 × 2

					1	0	•	1	0			
×			1	0	1	0	•	0	1	0		
=	0	1	1	0	0	1	•	1	0	1	0	0

Floating Point Representation

$$\text{sign} \cdot |\text{mantissa}| \cdot base^{\text{exponent}}$$

- regular mantissa = 1.xxxxxx
- denormal numbers get as close to zero as possible: mantissa = 0.xxxxxx with min exponent
- IEEE FP Standard: base=2, single, double, extended widths
- Computing in Space: choose widths of fields + choose base
- Tradeoff:
  - **Performance**: small widths, larger base, truncation.
  - versus **Accuracy**: wide, base=2, round to even.
- Disadvantage: Floating Point arithmetic units tend to be very large compared to Integer/Fixed Point units.

Arithmetic takes Space on the DFE

- Addition/subtraction:
  - ~1 logic cell/bit for fixed point,
    while it takes hundreds of logic cells per floating point op
- Multiplication: Can use MULT blocks
  - 18x25bit multiply on Xilinx Vertex6
  - Number of MULTs depends on total bits (fixed point) or mantissa bitwidth (floating point)

Approximate space cost models

	Floating point: dfeFloat(E, M)		**Fixed point: dfeFix(I, F, *TWOSCMP*)**	
	MULTs	**LUTs**	**MULTs**	**LUTs**
Add/subtract	0	$O(M \times \log_2(E))$	0	I+F
Multiply	$O(\text{ceil}(M/18)^2)$	O(E)	$O(\text{ceil}((I+F)/18)^2)$	0
Divide	0	$O(M^2)$	0	$O((I+F)^2)$

I = Integer bits, F = Fraction bits. E = Exponent bits, M = Mantissa Bits

MULT usage for N x M multiplication

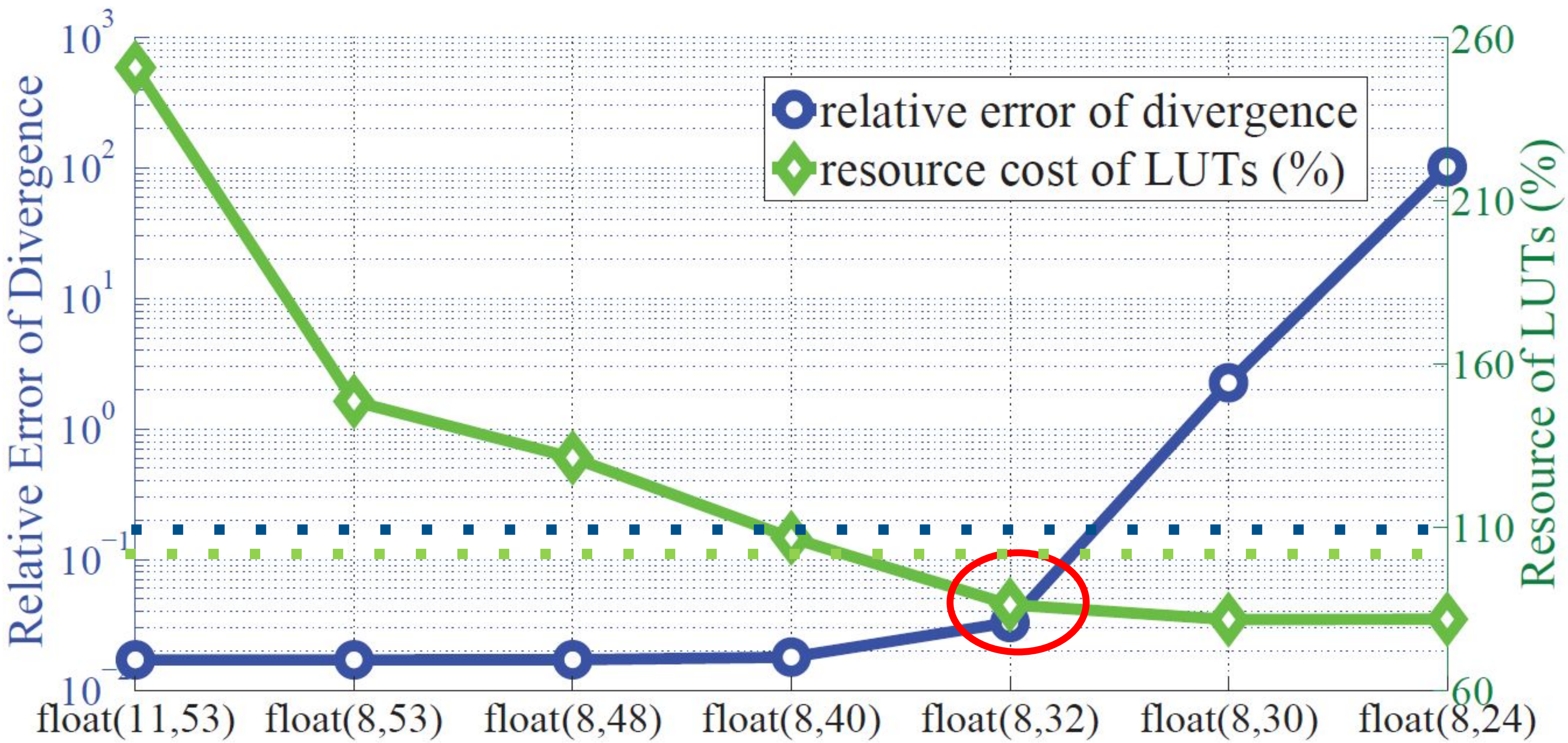
M

N

Bits	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54
18	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3
20	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
22	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
24	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
26	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
28	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
30	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
32	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
34	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
36	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
38	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
40	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
42	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
44	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
46	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
48	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
50	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
52	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
54	3	4	4	4	6	6	6	6	6	7	7	7	7	9	9	9	9	9	10

What about error vs area tradeoffs

- Bit accurate simulations for different bit-width configurations.



[L. Gan, H. Fu, W. Luk, C. Yang, W. Xue, X. Huang, Y. Zhang, and G. Yang, Accelerating solvers for global atmospheric equations through mixed-precision data flow engine, FPL2013]

Finally

- FPGAs are coming
- FPGA (hardware) programming requires a different mindset than software programming.
- Algorithmic differences
- Numerical differences