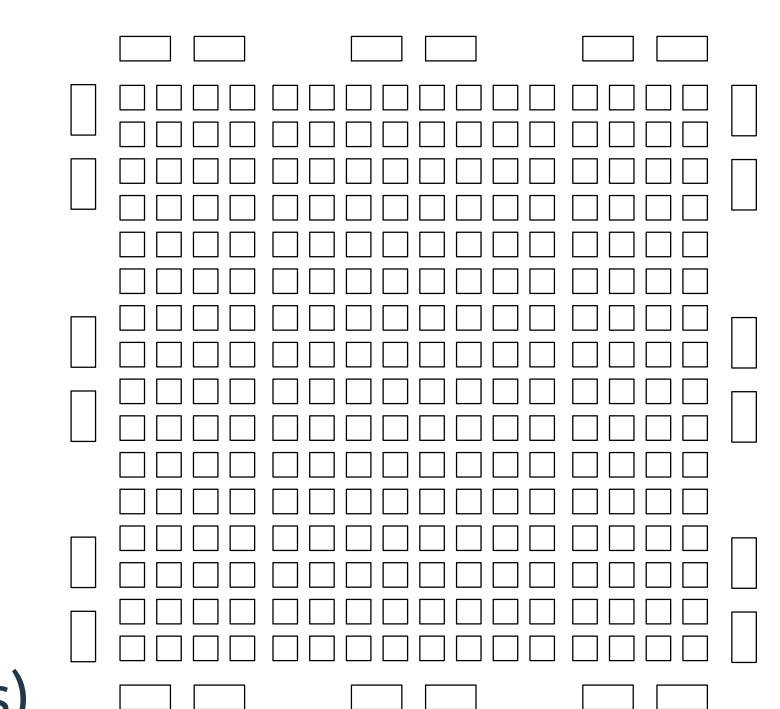
# PROGRAMMING FPGAS WITH DPC++

Adrian Jackson

a.jackson@epcc.ed.ac.uk

# What is an FPGA?



- Field programmable gate array
  - Programmable logic blocks
  - Memory units
  - Digital signal processing blocks
  - Interconnect
  - Other specialised pre-configured units (i.e. soft cores)
- Spatial architecture rather than instruction set architecture
  - Basic building blocks configurable to create your own processor
  - Reprogrammable as required
- Specialised processors design for individual applications rather than general purpose processor

# Instruction set architecture

- Processor implements pre-defined instruction set
  - Implementation provides functional units for each (set of ) instruction(s)
  - Implementation defines fix control flow for each (set of instruction(s)
  - Instruction flow architecture
  - Pipelining of instructions

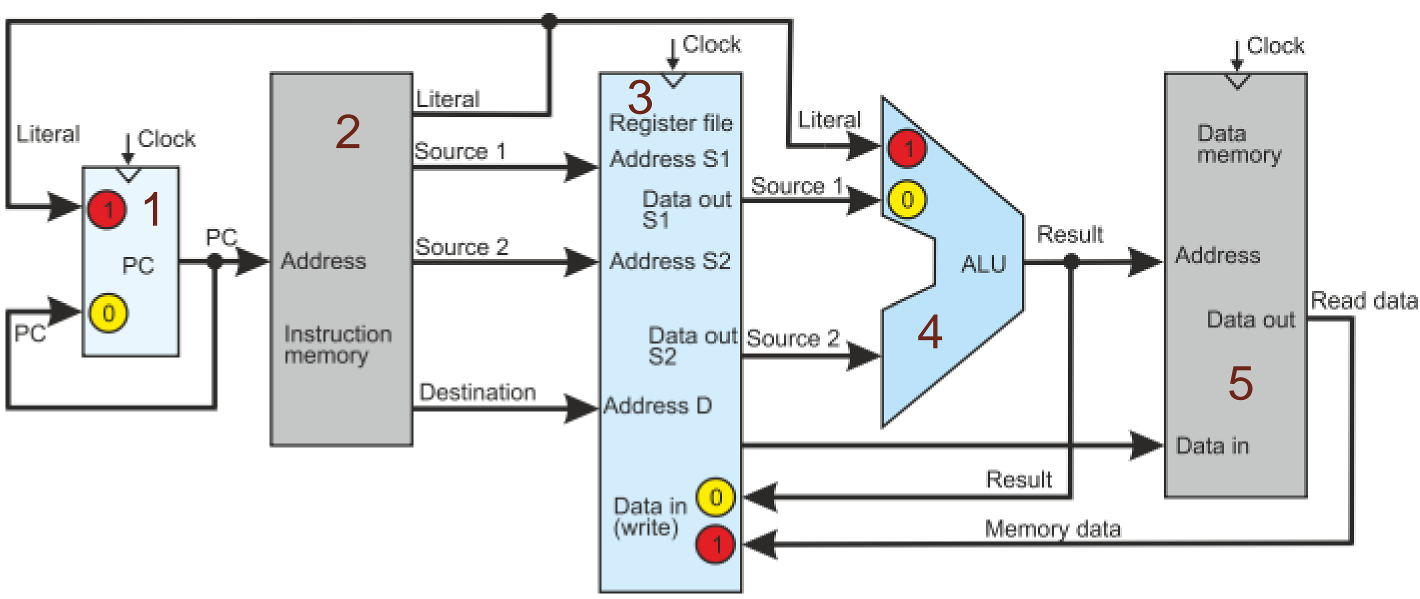
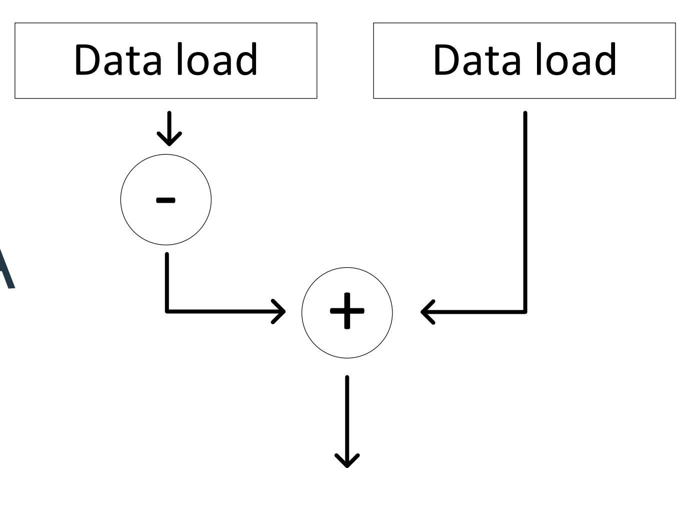
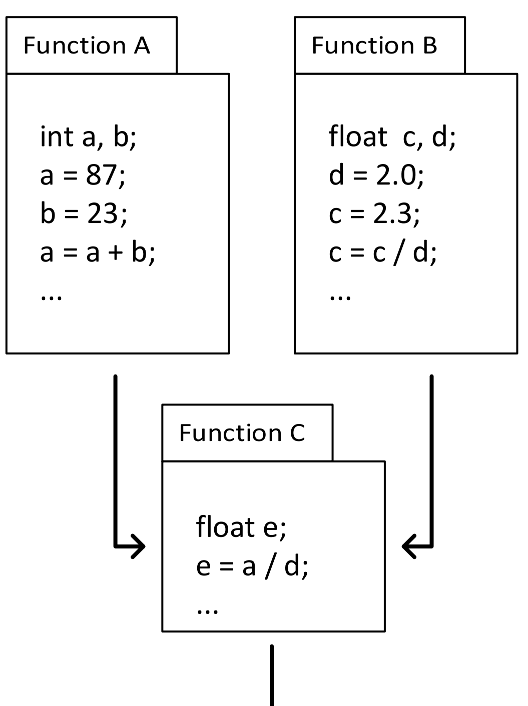



Image from http://alanclements.org/simple%20isa.html

# Dataflow architecture

- Individual program region encoded as part of FPGA
- Program region connected together
- Data flows through the hardware
  - Can recirculate as required
- All parts of the FPGA can be active
- Programmed on run
  - Hardware configuration pre-built
- Spatial computing



# Why FPGAs?

- Processor design tools
- Enable removing unused hardware components
  - Enable 100% usage of hardware
- Sidestep ISA limitations
  - i.e. dependency chains for vectorization
  - i.e. instruction issue bottlenecks
  - i.e. register pressure
  - i.e. fixed width precisions
  - etc...
- High memory bandwidth and data connectivity
  - Streaming data processing etc...

# Why not FPGAs?

- Programming is hard
  - Low level
  - Multiple dimensional optimisation problem
- Portability isn't straight forward
- Large program performance can be limited
  - FPGA footprint is key to efficiency
  - Reuse of space can fit large programs onto the device
- Fixed hardware blocks need to be used to get maximum performance
- Dynamic reprogramming possible, but dangerous (glitch)
- Programming can only be done when powered on
  - But program maintained once loaded

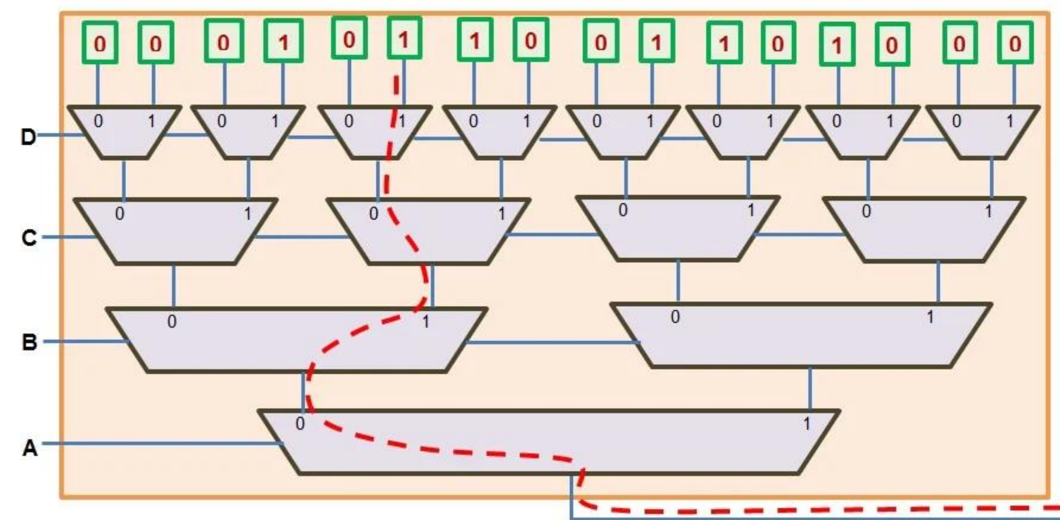
# FPGA components

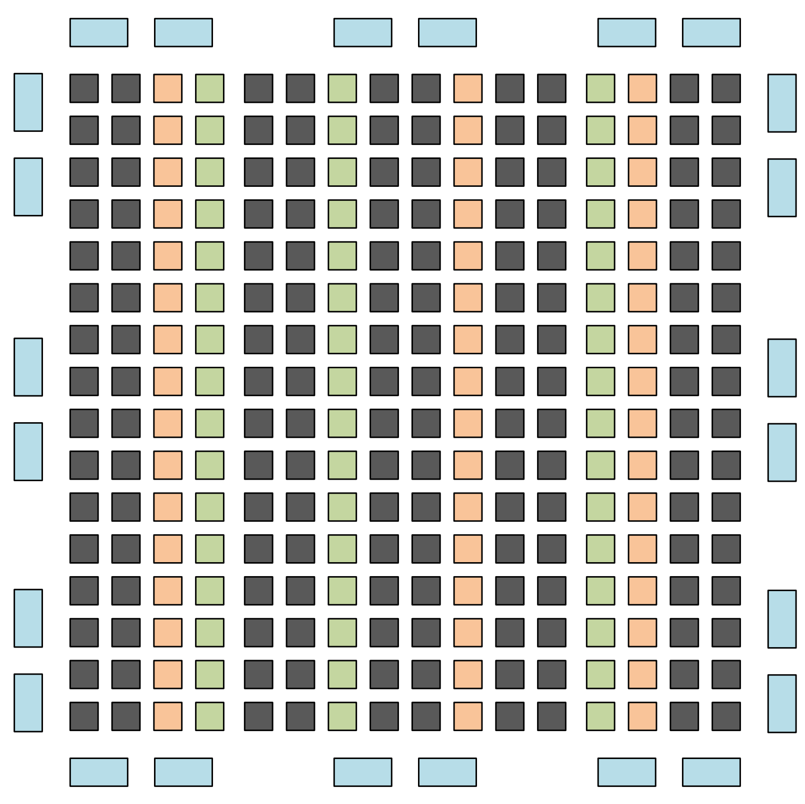
- Programmable logical blocks
  - Lookup tables
  - Adders to combine
  - Mimic hard gates

carry in
clk
a
b
c
d
3-LUT
3-LUT
FA
mux
mux
in
out
DFF
mux
out
logic cell
carry out
clk

**Truth Table**

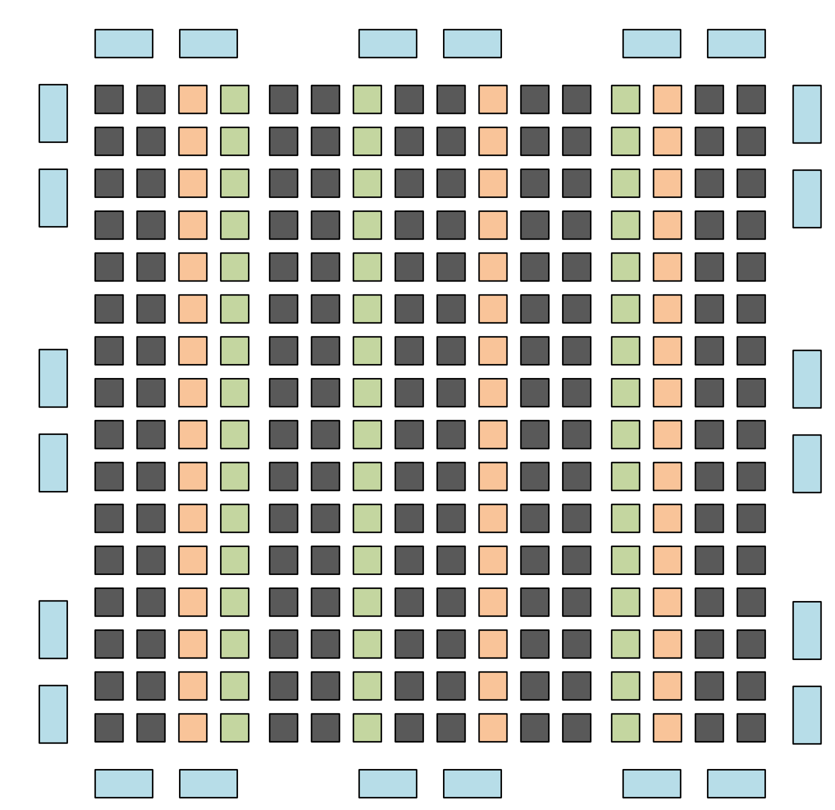
| Inputs | | | | Output |
|---|---|---|---|---|
| A | B | C | D | Y |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |



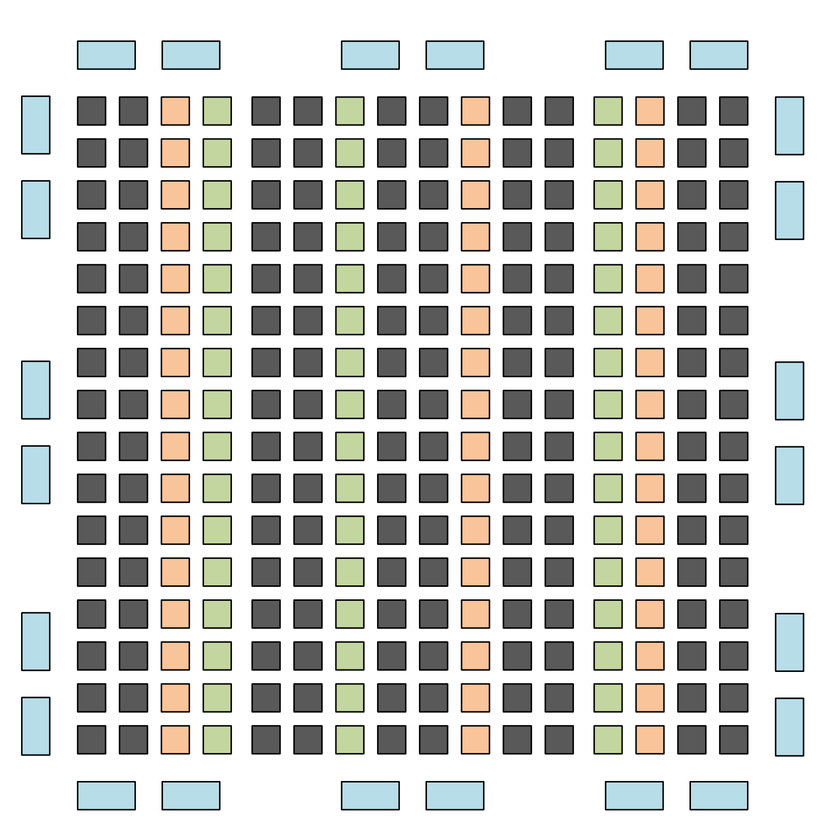
# FPGA components

- Local memory units
  - Block RAM
  - Multiple connections (Ports) possible
  - Configurable widths
- DSP blocks
  - Constrained FPUs
  - Multiplexers and ALUs



# FPGA components

- Interconnects
  - Routing connections between different components
  - Configurable
  - High overall bandwidth
- Connection blocks
  - PCI connections
  - Memory controllers
- Hard cores
  - Embedded full hardware implementations
  - i.e. Intel Stratix has:
    - Quad-core 64-bit ARM Cortex-A53 processor
    - PCIe Gen1/Gen2/Gen3 complete protocol stack
    - 10/25/100 Gbps Ethernet MAC
    - DDR4/DDR3/LPDDR3 hard memory controller



# Programming FPGAs

- Traditional programming
  - Map program to ISA: Compiler
  - Language/compiler specification maps language features to ISAs
  - Optimisation done on specific ISA constraints and specific processor constraints
- FPGA programming
  - Deciding how to configure the hardware for the program you want to run
  - Decide what component implementations to put where on the FPGA (placement and routing)
  - Connect up all the external resources (memory and networks)
  - Create FPGA bitstream to program the device

# Traditional FPGA programming

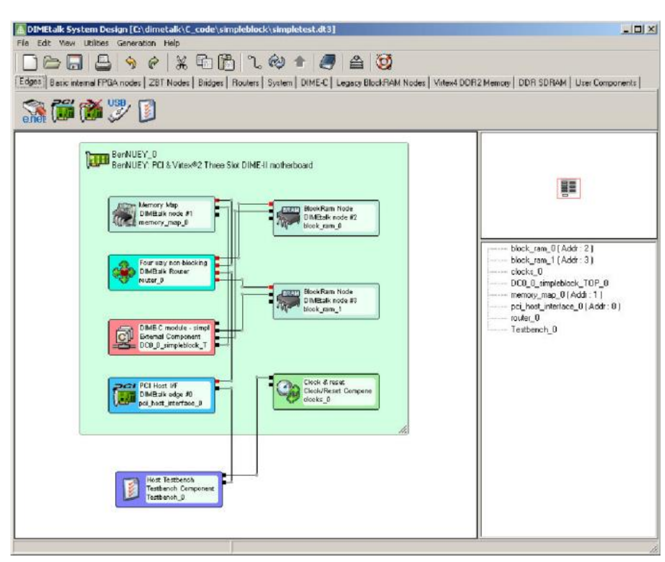
- Low level design tools
  - VHDL, Verilog
  - Write LUTs for specific hardware
  - Create layout and connection maps
  - Synthesis
  - Placement
  - Routing

```
module tb ();
  reg a, b, c;
  wire y;

  design dut (.a(a), .b(b), .c(c), y(y)

  // apply input sequence

  initial
    begin

      a = 0; b = 0; c = 0 ; # 10;
      a = 0; b = 0; c = 1 ; # 10;
      a = 0; b = 1; c = 0 ; # 10;
      a = 0; b = 1; c = 1 ; # 10;
      a = 1; b = 0; c = 0 ; # 10;
      a = 1; b = 0; c = 1 ; # 10;
      a = 1; b = 1; c = 0 ; # 10;
      a = 1; b = 1; c = 1 ; # 10;

    end
endmodule
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity signed_adder is
6    port
7    (
8      aclr : in    std_logic;
9      clk  : in    std_logic;
10     a    : in    std_logic_vector;
11     b    : in    std_logic_vector;
12     q    : out   std_logic_vector
13   );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert (a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27     begin
28       if (aclr = '1') then
29         q_s <= (others => '0');
30       elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32       end if; -- clk'd
33     end process;
34
35 end signed_adder_arch;
```

# Higher level programming

- C to Gates
  - C to Register Transfer Level common tool for productivity
- Automated tools to take higher level languages to bitstream
  - Use compiler to convert program into LUTs and DSP blocks
  - Undertake placement, routing, and synthesis
  - Highly unconstrained problem
  - Still requires user help and input
- OpenCL took this approach
  - Basis for other high-level tools



# OneAPI DPC++

- DPC++ follows on from OpenCL
  - Uses SYCL to port to FPGA
  - Integrates Intel's (Altera's) Quartus FPGA tools
- Convert application code into SYCL
- Convert SYCL to HDL using HLS
- Synthesis HLS and build bitstream
- Run the code and program the FPGA at runtime

```
#include <CL/sycl.hpp>
using namespace sycl;

static const int N = 8;

int main(){
  queue q;
  std::cout << "Device: " <<
q.get_device().get_info<info::device::name>() <<
std::endl;

  int *data = malloc_shared<int>(N, q);
  for(int i=0; i<N; i++) data[i] = i;

  q.parallel_for(range<1>(N), [=] (id<1> i){
    data[i] *= 2;
  }).wait();

  for(int i=0; i<N; i++) std::cout << data[i] <<
std::endl;
  free(data, q);
  return 0;
}
```

# Compile cycle

- Synthesis (routing and placement) is slow
  - Hours
- Simulated/emulate synthesis can be used to test build and compile phase
  - Enables quick compile/development loop
  - Does not necessarily pick up all hardware issues

| Device Image Type | Time to Compile | Description |
| --- | --- | --- |
| **FPGA Emulator** | Seconds | The FPGA device code is compiled to the CPU. Use the Intel® FPGA Emulation Platform for OpenCL™ software to verify your SYCL code's functional correctness. |
| **FPGA Simulator** | Minutes | The FPGA device code is compiled to the CPU. Use the Questa*-Intel® FPGA Edition simulator to debug your code. |
| **Optimization Report** | Minutes | The FPGA device code is partially compiled for hardware. The compiler generates an optimization report that describes the structures generated on the FPGA, identifies performance bottlenecks, and estimates resource utilization. |
| **FPGA Hardware Image** | Hours | Generates the real FPGA bitstream to execute on the target FPGA platform. |

# Compilation

- DPC++ standard tools

```
dpcpp -fintelfpga -Xshardware my_source_code.cpp
dpcpp -fintelfpga -Xsboard=intel_a10gx_pac my_source_code.cpp
```

- Emulate rather than compile fully

```
dpcpp -fintelfpga my_source_code.cpp
```

- Optimisation report

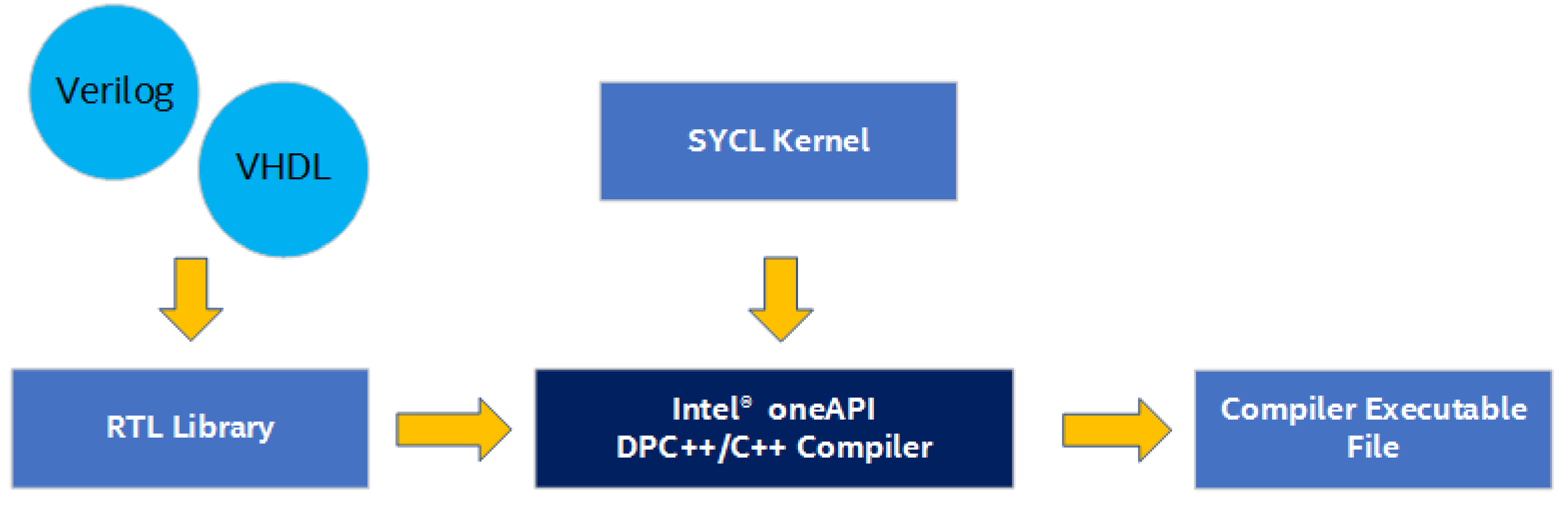
```
dpcpp -fsycl-link=early my_source_code.cpp
```

- Simulate rather than emulate

```
dpcpp -fintelfpga fpga_compile.cpp -Xssimulation -Xsboard=intel_s10sx_pac:pac_s10
```

# Mixing sources

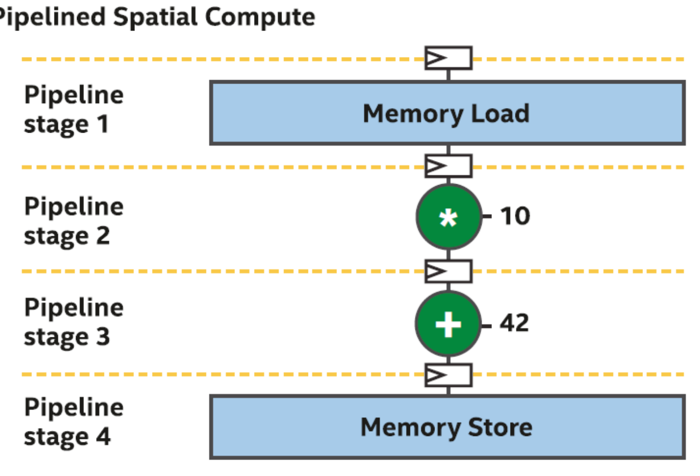
- Possible to include a range of different implement outputs




  - Simply call the function from your code

# Program design

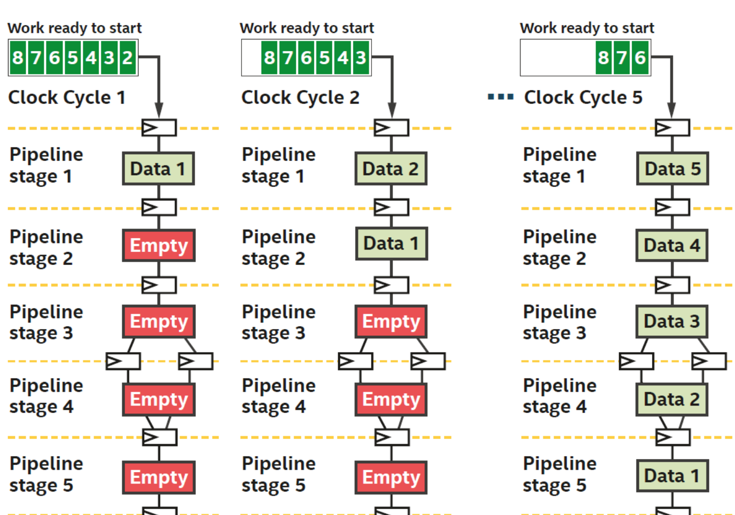
- Pipeline parallelism is the key approach
  - Aim for all hardware parts to be active all the time (after startup)
  - High throughput (reduced clock speed)
  - Large aggregate memory bandwidth
  - External memory access can reduce performance




Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X. (2021). Programming for FPGAs. In: Data Parallel C++. https://link.springer.com/content/pdf/10.1007/978-1-4842-5574-2.pdf

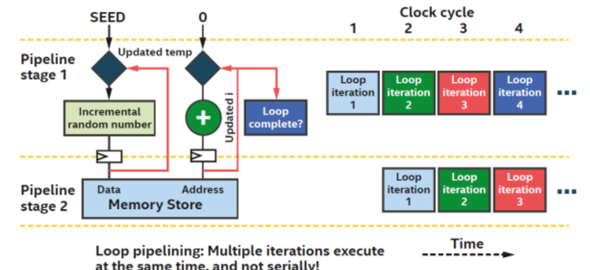
# Program design

- Spatial parallelism
  - Program blocks mapped to hardware
- Memory/data parallelism
  - Direct memory configuration



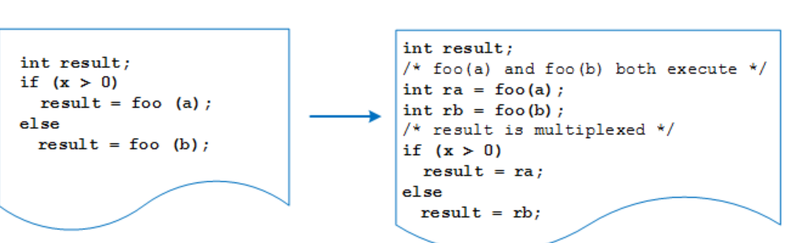

- Both loop and vectorisation type optimisation possible
  - FPGA is often considered for spatial parallelism, but loop or vectorisation styles can also be mapped to the FPGA
  - Vectorisation may not provide the most efficient method (high space usage)

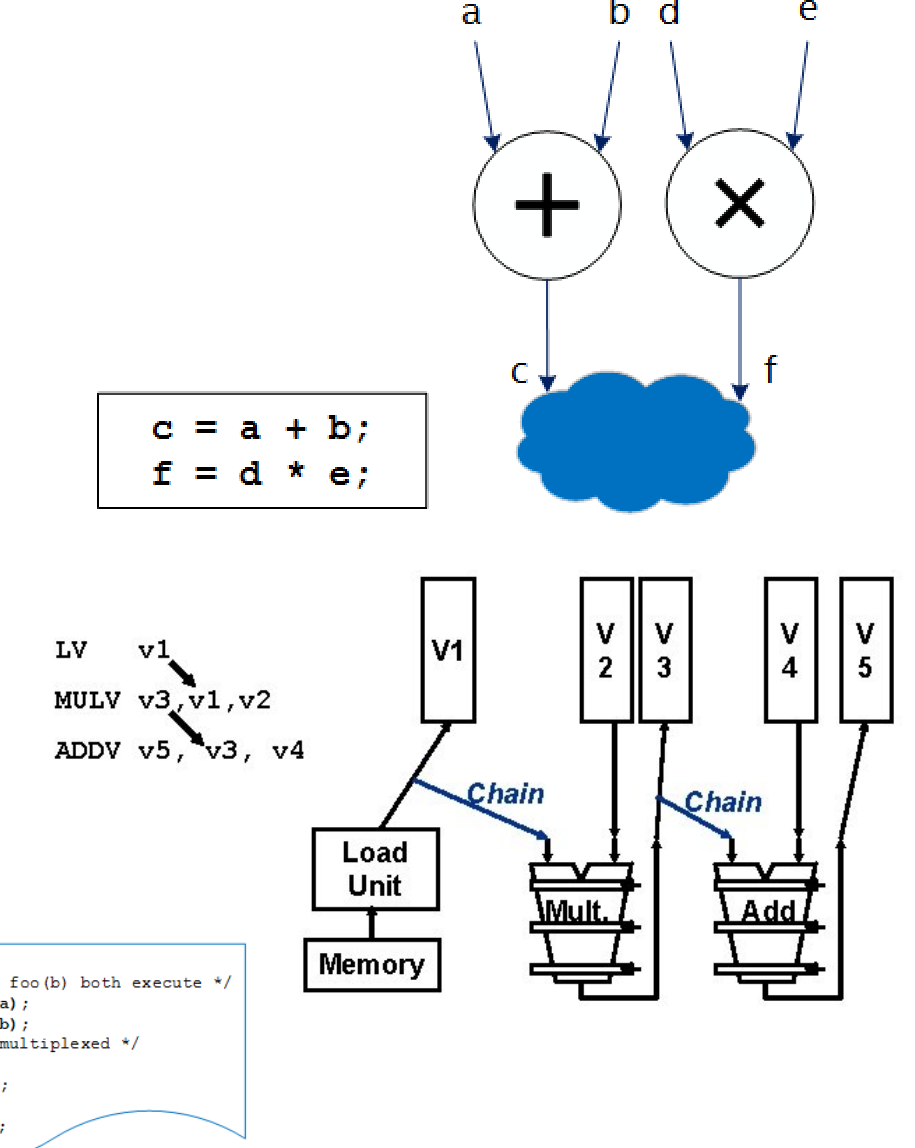
```
int a = 0;
for (int i=0; i < size; i++) {
  a = a + i;
}
```



# Program design

- Instruction level parallelism
  - Done at compile time
  - Trade off with space usage
- Vectorisation
  - Old school pipeline vectorization
- Loop unrolling
  - Spread loop iterations across hardware
- Conditional unbundling



# Program design

- Exposing parallelism is wider than other systems
  - Functions
  - Loops
  - Vectors
  - Whole call stacks
  - etc...
- Data costs still as important
  - Reuse
  - Locality
  - etc...

# DPC++

- Keeping pipelines full key for performance
- ND-range kernels allow compiler to queue and dispatch these

```
h.parallel_for({16,16,16}, [=](auto I) {
  output[I] = generate_random_number_from_ID(I);
});
```

- Standard loops allow iterations to be queued and dispatched
  - Unrolling allows further parallelisation
  - Space trade off vs throughput/pipeline depth

```
#pragma unroll 4
for(i = 0 ; i < 20; i++){
  a[i] += 1;
}
```

# Dependencies

```
#include <CL/sycl.hpp>
using namespace sycl;
static const int N = 4;
int main(){
  queue q;
  int *d1 = malloc_shared<int>(N, q);
  int *d2 = malloc_shared<int>(N, q);
  for(int i=0; i<N; i++) { d1[i] = 10; d2[i] = 10; }

  auto e1 = q.parallel_for(range<1>(N), [=] (id<1> i){
    d1[i] += 2;
  });

  auto e2 = q.parallel_for(range<1>(N), [=] (id<1> i){
    d2[i] += 3;
  });

  q.parallel_for(range<1>(N), {e1, e2},  [=] (id<1> i){
    d1[i] += d2[i];
  }).wait();

  for(int i=0; i<N; i++) std::cout << d1[i] << std::endl;

  free(d1, q);
  free(d2, q);
  return 0;
}
```

# DPC++

- Memory types
  - Functions for different memory functionality

```
group_local_memory_for_overwrite()
struct State {
 [[intel::fpga_memory]] int array[100];
 [[intel::fpga_register]] int reg[4];
};
cgh.single_task<class test>([=] {
  struct State S1;
  [[intel::fpga_memory]] struct State S2;
  // some uses
});
```

  - Different memory device types possible as well

THE UNIVERSITY of EDINBURGH

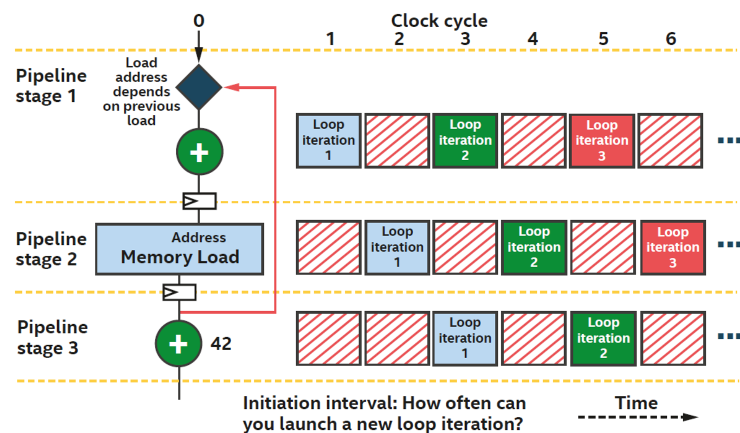
# DPC++

- Ensuring data types match across assignments helps performance
- Variable precision support

| Data Type | Header File | Description |
| --- | --- | --- |
| ac_int | <sycl/ext/intel/ac_types/ac_int.hpp> | Arbitrary-precision integer support |
| ac_fixed | <sycl/ext/intel/ac_types/ac_fixed.hpp> | Arbitrary-precision fixed-point number support |
| ac_fixed_math | <sycl/ext/intel/ac_types/ac_fixed_math.hpp> | Support for some non-standard math functions for arbitrary-precision fixed-point data types. |
| ac_complex | <sycl/ext/intel/ac_types/ac_complex.hpp> | Complex number support |
| ap_float | <sycl/ext/intel/ac_types/ap_float.hpp> | Arbitrary-precision floating-point number support |
| ap_float_math | <sycl/ext/intel/ac_types/ap_float_math.hpp> | Support for commonly used exponential, logarithmic, power, and trigonometric functions with ap_float type |

# Performance issues

- Data dependencies can stall pipelines
  - Occupancy metric used to quantify how much of the hardware used
  - Aiming for 100% usage and 100% occupancy
- Full usage may lead to network issues
  - Reduced clock speed or stalls
- Automatic memory types may not be ideal
  - Chosen types can be tuned for performance
- Memory banking can also be tuned
- As can access coalescing




```
T* ptr = malloc_device<T>(1024, Queue);
...
cgh.single_task<class DeviceAnnotation>([=]() {
  Ptr[0] = 42;
  device_ptr<T> DevicePtr(Ptr);
  DevicePtr[1] = 43;
});
```

THE UNIVERSITY of EDINBURGH

# Performance issues

- Automatic transfer (host<->device) can be implemented for you
  - DPC++ can tagged buffer read-only or write-only,
  - Concurrent PCIe transfer possible
  - Double buffering to allow asynchronous transfers
- Variable scoping
  - Minimising variable scoping extents can reduce networking and memory usage
- Loop optimisations
  - Collapsing
  - Dependency reduction
  - Exit clauses and conditionals
  - Etc…

# Loop performance example

```
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  accessor A(A_buf, cgh, read_only);
  accessor B(B_buf, cgh, read_only);
  accessor Result(Result_buf, cgh, write_only);
  cgh.single_task<class unoptimized>([=]() {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        sum += A[i * N + j];
      }
      sum += B[i];
    }
    Result[0] = sum;
  });
});
```

https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design.html

```
constexpr int N = 128;
 queue.submit([&](handler &cgh) {
   accessor A(A_buf, cgh, read_only);
   accessor B(B_buf, cgh, read_only);
   accessor Result(Result_buf, cgh, write_only);
   cgh.single_task<class optimized>([=]() {
     int sum = 0;
     for (int i = 0; i < N; i++) {
       // Step 1: Definition
       int sum2 = 0;
       // Step 2: Accumulation of array A values
       // for one outer
       // loop iteration
       for (int j = 0; j < N; j++) {
          sum2 += A[i * N + j];
       }
       // Step 3: Addition of array B value for an
       // outer loop iteration
       sum += sum2;
       sum += B[i];
     }
     Result[0] = sum;
   });
 });
```

# FPGA testbed

- EPCC is assembling an FPGA tested
  - Funded by the UKRI Excalibur programme
  - https://fpga.epcc.ed.ac.uk/
- Mixture of FPGAs to try
  - Intel Stratix…
  - Xilinx Alveo and Versal
  - Others to come
- Direct FPGA to FPGA networking
- Host resources and software environments
- Get in touch for access