Hardware Software Codesign with FPGAs

Instructor:

Prof. Jim Plusquellic

Text:

- A Practical Introduction to Hardware/Software Codesign, 2nd Edition", Patrick Schaumont, Springer, ISBN 978-1-4614-3736-9, 978-1-4614-3737-6 (eBook)

References:

- Hardware/Software Co-Design - Principles and Practice"
  J. Staunstrup and W. Wolf, Kluwer Academic Publishers, 1997.
  ISBN: 0792380134
- "Co-Design for System Acceleration A Quantitative Approach"
  Nadia Nedjah, Luiza de Macedo Mourelle, 2007.
  ISBN: 978-1-4020-5545-4
- "Embedded System Design, A Unified Hardware/Software Introduction"
  Frank Vahid and Tony Givargis, 2002.
  ISBN: 978-0-471-38678-0

Face-to-face Website: http://www.ece.unm.edu/jimp/codesign

Introduction

Relevant Conferences/Symposia on Codesign:

- Conference on Formal Methods and Models for Codesign (MEMOCODE)

- CODES+ISSS: The premier conference for System-Level Design, Embedded Systems Week

  The CODES+ISSS Conference is the merger of two major international symposia on hardware/software codesign and system synthesis.

- DAC: Design Automation Conference

- ASP-DAC: Asia South Pacific Design Automation Conference

- CASES: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems

- ICCAD: International Conference on Computer Aided Design

The Nature of Hardware and Software

What is H/S Codesign (Prof. Schaumont's definition):

Hardware/Software Codesign is the partitioning and design of an application in terms of fixed and flexible components

Other definitions

- HW/SW Codesign is a design methodology supporting the **concurrent** development of hardware and software (cospecification, codevelopment and coverification) in order to achieve *shared functionality and performance goals* for a combined system

- HW/SW Codesign means *meeting system level objectives* by exploiting the **synergism** of hardware and software through their concurrent design

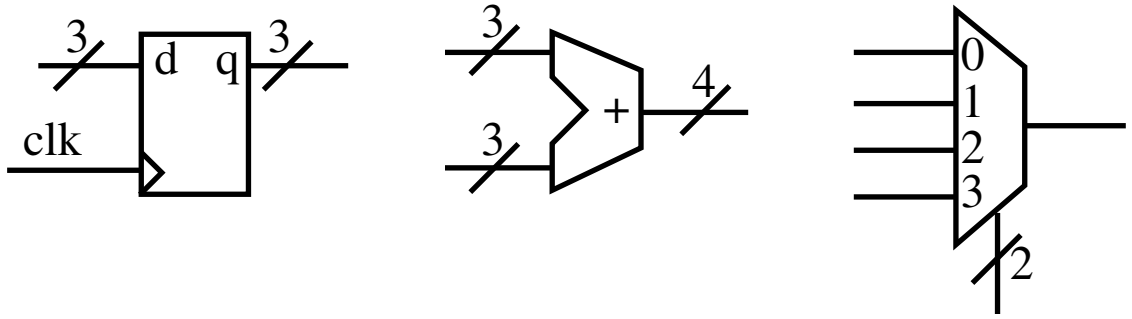
  Giovanni De Micheli and Rajesh Gupta, "Hardware/Software Co-design", IEEE Proceedings, vol. 85, no.3, March 1997, pp. 349-365

- Codesign is the concurrent development of hardware and software

Hardware

We assume **hardware** refers to *single-clock synchronous* digital circuits

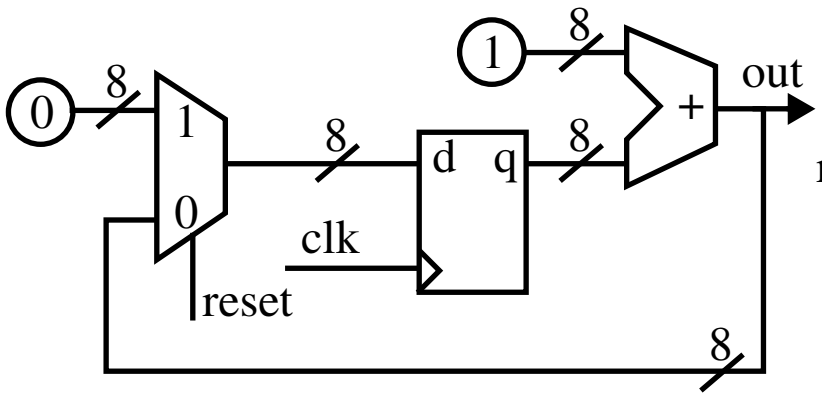
Hardware is realized by word-level combinational and sequential components, such as registers, MUXs, adders and multipliers



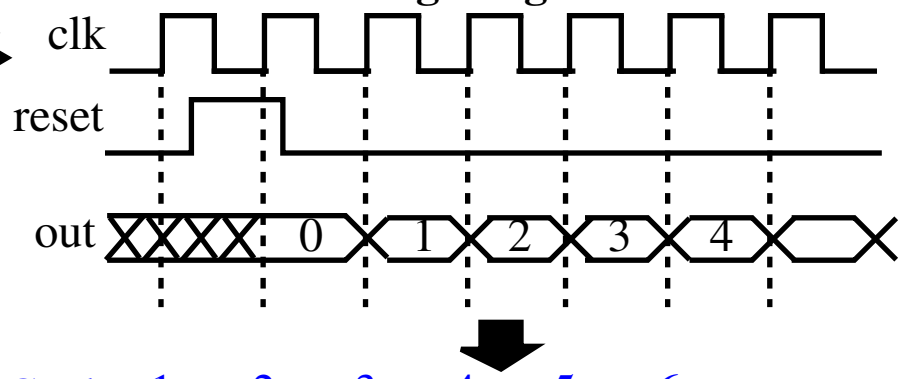
Cycle-based word-parallel hardware modeling is called **register-transfer-level** (RTL) modeling

RTL refers to a description in which the design is **abstracted** as a set of operations performed on data as it is transferred between registers

Hardware



Timing Diagram



This model allows us to **ignore** all events in between steps, and use a set of numbers to represent behavior, i.e. waveforms not needed →

Cycle	1	2	3	4	5	6
Reset	0	1	0	0	0	0
out	x	0	1	2	3	4

Bear in mind that this is a very *simplistic* treatment of actual hardware

We ignore advanced circuit styles including *asynchronous hardware*, *dynamic logic*, *multi-phase clocked hardware*, etc.

The cycle-based model is **limited** because it does not model *glitches*, *race conditions* or events that occur within clk cycles

However, it provides a convenient abstraction for a designer who is mapping a behavior, e.g., an algorithm, into a set of discrete steps

Software

We assume **software** refers to a *single-thread sequential* program written in C or assembly program

Programs will be shown as listings, e.g., **Listing 1.1** C example

```
1 int max;
2
3 int findmax(int a[10]) {
4    unsigned i;
5    max = a[0];
6    for (i = 1; i < 10; i++)
7       if (a[i] > max) max = a[i];
8 }
```

Software

Listing 1.2 ARM assembly example

```
                ldr     r2,   .L10          "max = a[0];"
                ldr     r3,   [r0, #0]      "max = a[0];"
                str     r3,   [r2, #0]      "max = a[0];"
                mov     ip,   #1             "for loop"
.L7:
                ldr     r1,   [r0, ip, asl #2] "scale i"
                ldr     r3,   [r2, #0]       "read a[i]"
                add     ip,   ip, #1         "i++"
                cmp     r1,   r3             "a[i] > max"
                strgt   r1,   [r2, #0]       "cond. store"
                cmp     ip,   #9             "i < 10"
                movhi   pc,   lr            "cond. return"
                b       .L7                   "uncond. br"
.L11:
                .align  2
.L10:
                .word   max
```

Hardware and Software

Hardware and software are *modeled* using RTL and C programs

A designer creates a model, i.e., an abstract representation, of the system from a specification

For example, a VHDL behavioral description is a *model* of the hardware circuit, which consists of gates and wires

Similarly, a C program is a *model* of a set of micro-processor instructions, i.e., a binary

NOTE: C programs are rarely referred to as models, and instead many generally consider them as actual implementations

Models and programs are used as input to simulation and implementation tools

Hardware/software codesign uses both models (RTL) and programs (C) as descriptions of a system implementation

Hardware/Software Ambiguities

There are many examples of systems in which the distinction between hardware and software is not always crystal clear

For example:

- An FPGA is a hardware circuit that can be reconfigured

  The **program** for an FPGA is a **bitstream**, which is used to configure its logic function

  VHDL and verilog are used to generate software models, which in turn are translated into bitstreams that configure the hardware

- A **soft-core** is a processor configured into an FPGA's reconfigurable logic

  The soft-core can then be used to execute C programs

- A **digital signal processor** (DSP) has instructions which are optimized for signal-processing applications

  Efficient programs require detailed knowledge of the DSP HW architecture, which creates a strong relationship between software and hardware

Hardware/Software Ambiguities

- An **application-specific instruction-set processor** (ASIP) is a processor with a customizable instruction set

  Users can design customized instructions for the processor, which are later referenced in programs

- The **CELL processor**, used in the Playstation-3, contains one control processor and 8 slave-processors, interconnected through a high-speed on-chip network

  The software for a CELL is a set of 9 concurrent communicating programs and configuration instructions for the on-chip network

A common characteristic of all these examples is that creating the SW requires **intimate familiarity** the HW

Hardware vs. Software

Choosing between implementing an application in HW or implementing it in SW may seem like a no-brainer -- clearly writing software is easier!

Software is flexible, compilers are very fast, there are lots of libraries available and computing platform are cheap and plentiful

More importantly, it seems a waste of effort to design a new hardware system when it is easy just to purchase one, e.g., a desktop computer

So what are the drivers for custom hardware?

Let's consider two important metrics that are used to compare hardware and software

Performance and Energy Efficiency

Performance:

Expressed as the amount of work done per unit of time

Let's define a unit of *work* as the processing of 1 bit of data

Hardware vs. Software

The figure shows various embedded system cryptographic implementations in software and hardware that have been proposed over 2003-2008.

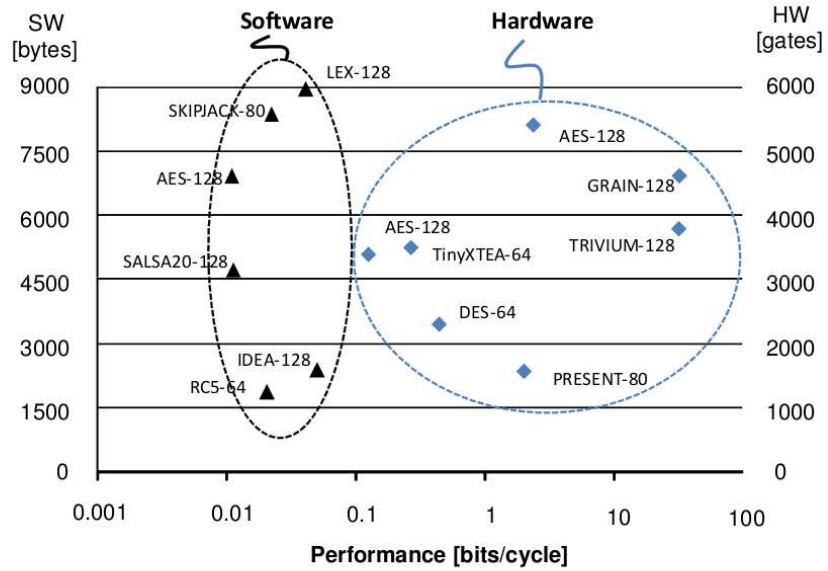


Fig. 1.4 Cryptography on Small Embedded Platforms

Performance in bits/cycle shown along x-axis shows that hardware has better performance than embedded processors (software)

Bear in mind that even though hardware executes more operations in parallel, high-end micro-processors have VERY high clk frequencies

Therefore, software may in fact *out-perform* dedicated hardware

Hardware vs. Software

Therefore, performance is **not** a very good metric to compare hardware and software, especially in resource-constrained environments such as IoT

A better metric (that is independent of clk frequency) is **energy efficiency**, i.e., the amount of useful work done per unit of energy

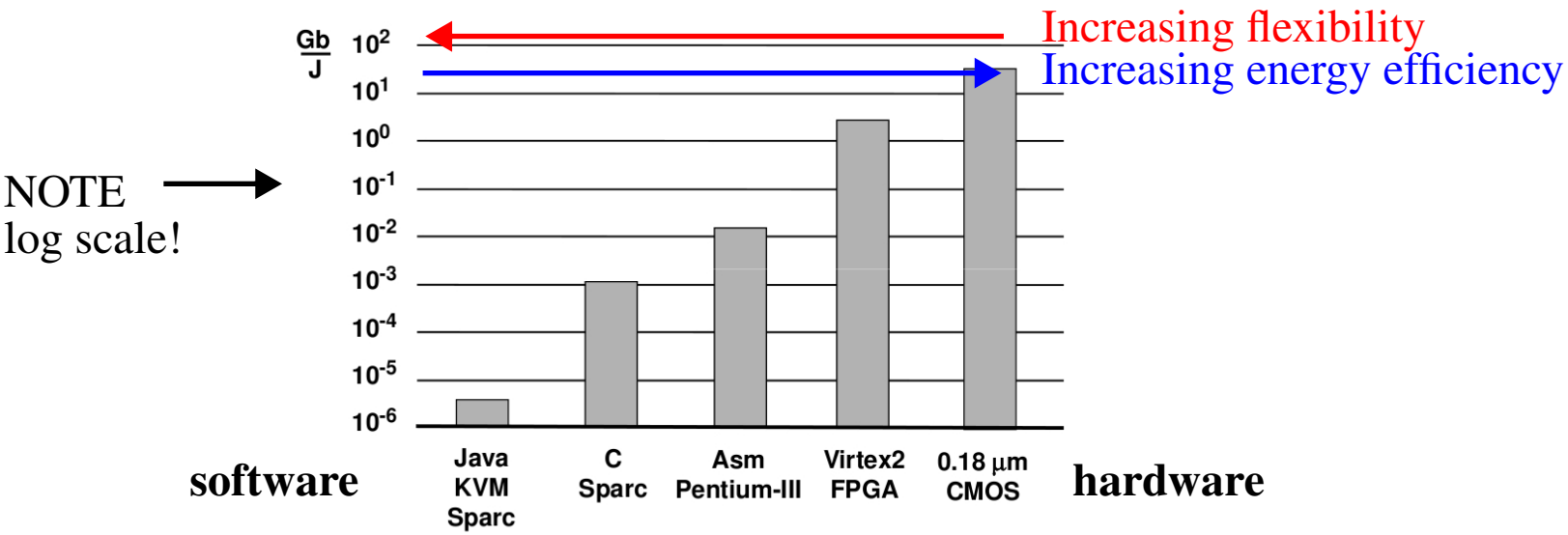


Fig. 1.5 Energy Efficiency

This graph shows energy consumption of an AES engine (encryption) on different architectures, with y-axis plotting Gigabytes per Joule of energy

This shows battery-operated devices would greatly benefit using less flexible, dedicated hardware engines

Hardware vs. Software

This is true b/c there is a **large overhead** associated with executing software instructions in the microprocessor implementation

- Instruction and operand fetch from memory
- Complex state machine for control of the datapath, etc.

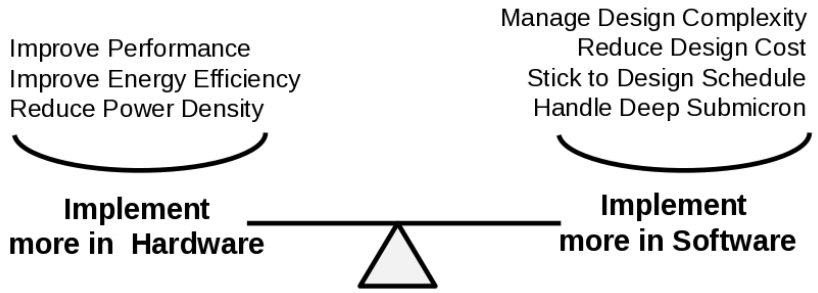
Also, specialized hardware architectures are usually also more efficient than software from a *relative performance perspective,* i.e., amount of useful work done per clock cycle

Flexibility comes with a significant energy cost -- one which energy optimized applications cannot tolerate

Therefore, you will **never find** a Pentium processor in a cell phone!

Hardware vs. Software

The complete picture of whether and how to implement a system is more complicated



Hardware or software present many trade-offs, some of which have conflicting objectives

Arguments in favor of increasing the amount of **hardware** (HW):

- **Performance and Energy Efficiency**:

  As indicated above, improvements in relative performance and energy efficiency is a big plus for hardware, especially battery-operated devices

  HW/SW codesign plays an important role in optimizing **energy-efficiency** by helping designers to decide which components of flexible SW should be moved into fixed HW

Hardware vs. Software

- **Power Densities**:

Further increasing clock speed in modern high-end processors as a performance enhancer has run-out-of-gas because of thermal limits

This is driven a broad and fundamental shift to increase **parallelism** within processor architectures

However, there is no dominant parallel computer architecture that has emerged as 'the best architecture' -- commercially available systems include

- Symmetric multiprocessors with shared memory
- Traditional processors tightly coupled with FPGAs as accelerator engines
- Multi-core and many-core architectures such as GPUs

Nor is there yet any universally adopted parallel programming language, i.e., code must be crafted differently depending on the target parallel platform

This forces programmers to be **architecturally-aware** of the target platform

Hardware vs. Software

Arguments for increasing the amount of software (SW):

- **Design Complexity**

  Modern electronic systems are extremely complex, containing multiple processors, large embedded memories, multiple peripherals and input-output devices

  It is generally difficult or impossible to design all components in fixed hardware

  On the other hand, software implementations running on processors can better handle complexity and additionally allows for updates and bug fixes

- **Design Cost**

  New chips are very expensive to design and fabricate

  Programmable architectures including processors and FPGAs are becoming more attractive because they can reused over multiple products or product generations

    System-on-Chip (SoCs) are good examples of this trend

Hardware vs. Software

- **Shrinking Design Schedules**

  Each new technology is more complex than the previous generation, and the move to the next generation happens more quickly

  For the designer, this means that each new product generation brings more work that needs to be completed in a shorter amount of time

  Shrinking design schedules require engineering teams to develop the HW and SW components of a system concurrently

  These trends also increase the attractiveness of software and microprocessor-based solutions

Finding the correct balance, while weighing in all these factors, is a complex problem

Instead, we will focus on optimizing metrics related to ***design cost*** and ***performance***

In particular, we will consider how adding hardware to a software implementation increases performance while weighing in the increase in design cost

The Hardware-Software Codesign Space

The proceeding discussion makes it apparent that there are a multitude of alternatives available for mapping an application to an architecture

The collection of all possible implementations is called the ***HW/SW codesign space***

The following figure represents the design space symbolically

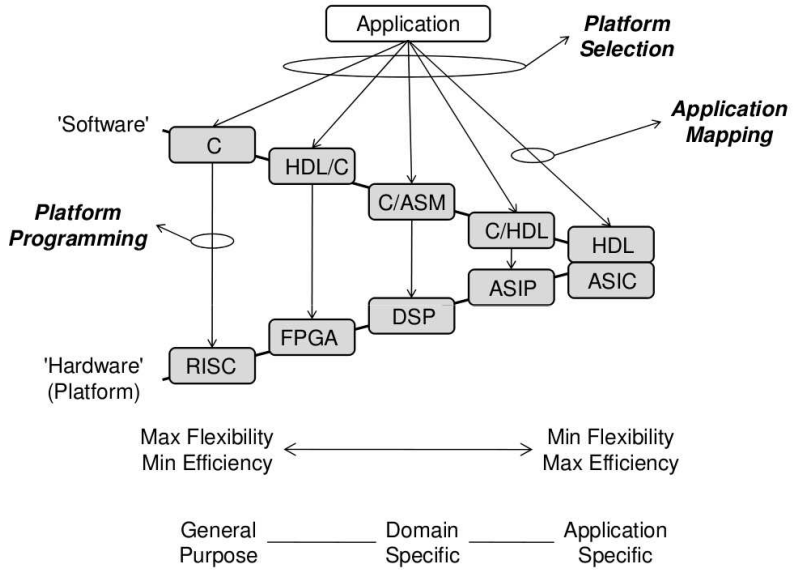
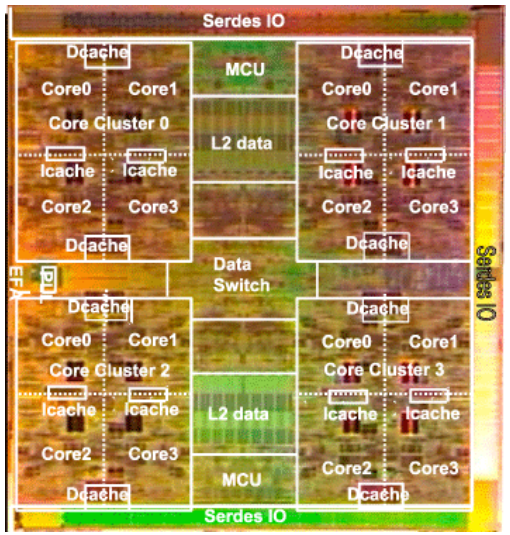


Fig. 1.7 The Hardware-Software Codesign Space

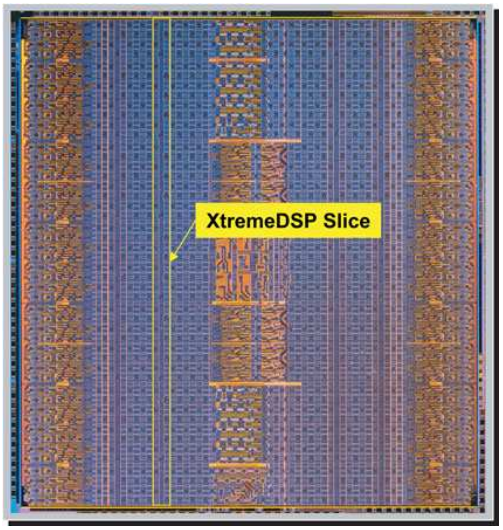
Platform Design Space: The objective of the design process is to map a *specification* onto a *target platform*

Examples Micrographs of Target Platforms

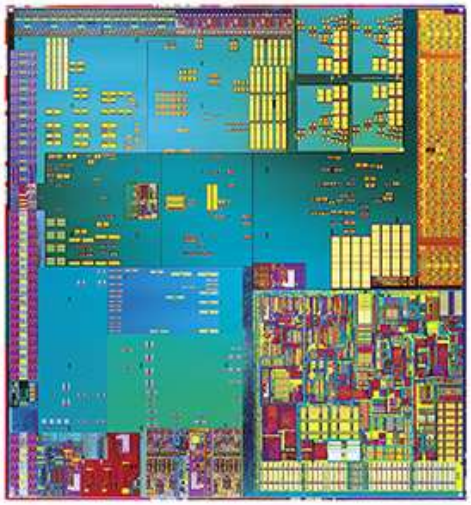
Microprocessor



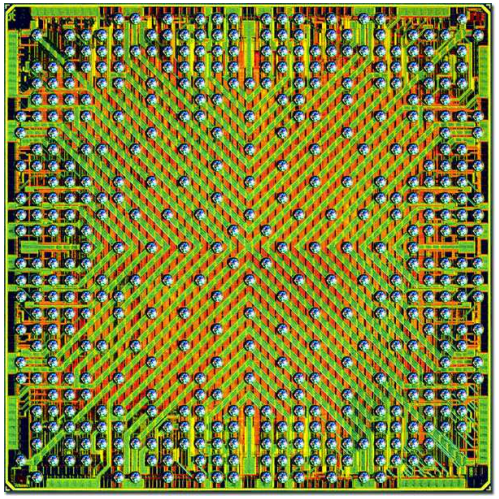
FPGA



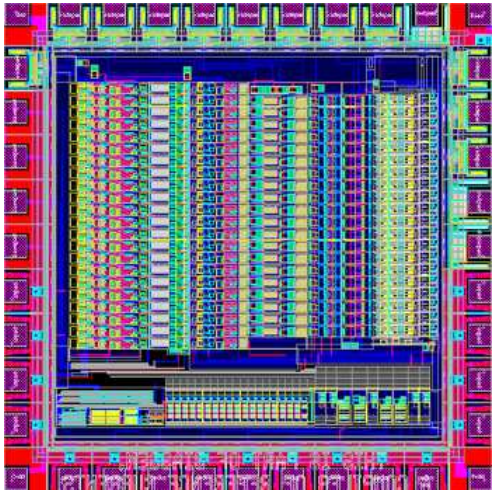
SoC



DSP

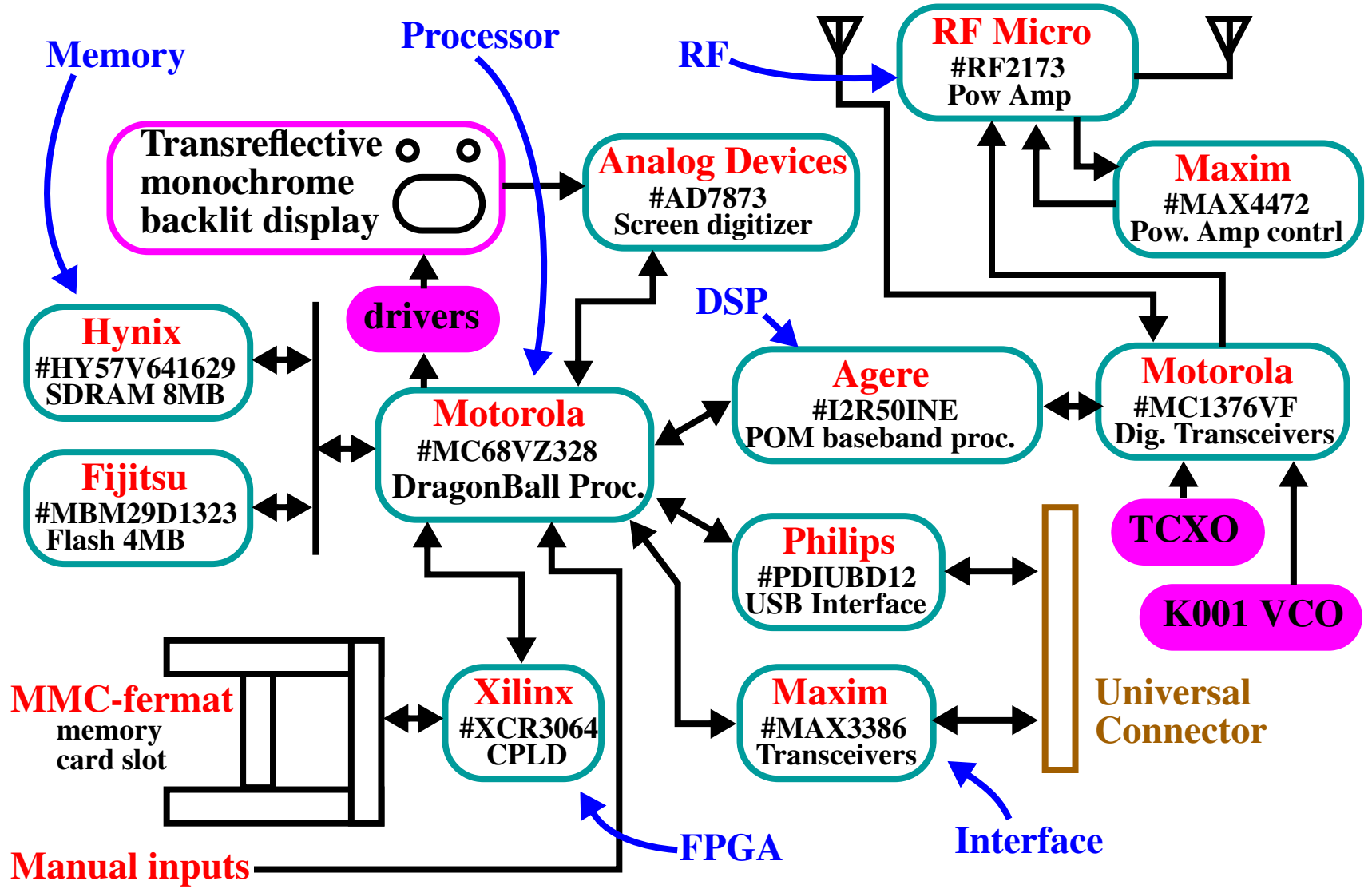


Microcontroller



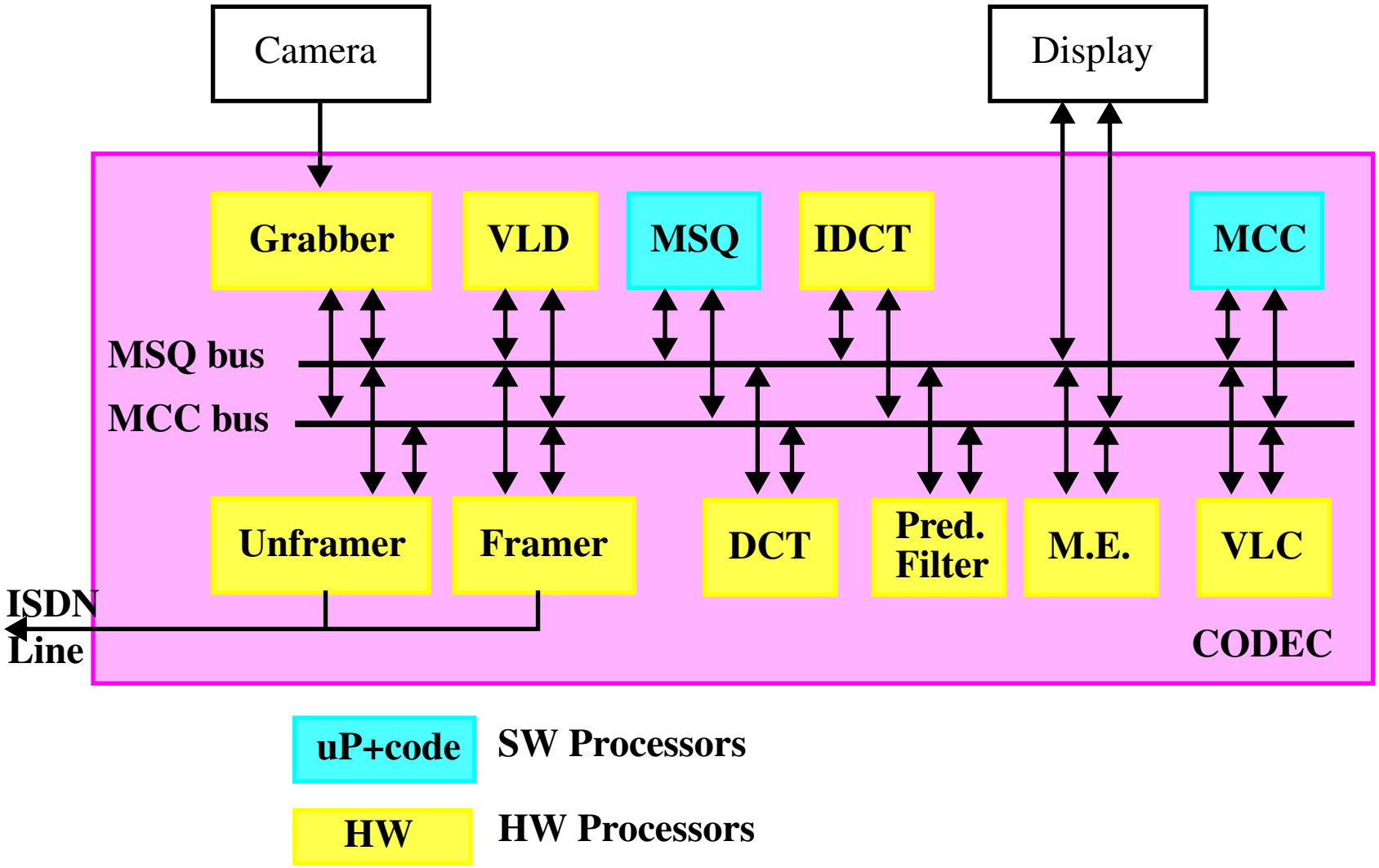
SoC Examples

Example System-on-Chip (SoC) with IP cores



Codesign Examples

Video Codec (H261)



The Hardware-Software Codesign Space

Each of the above platforms presents a trade-off between **flexibility** and **efficiency**

The wedge-shape of the diagram expresses this idea:

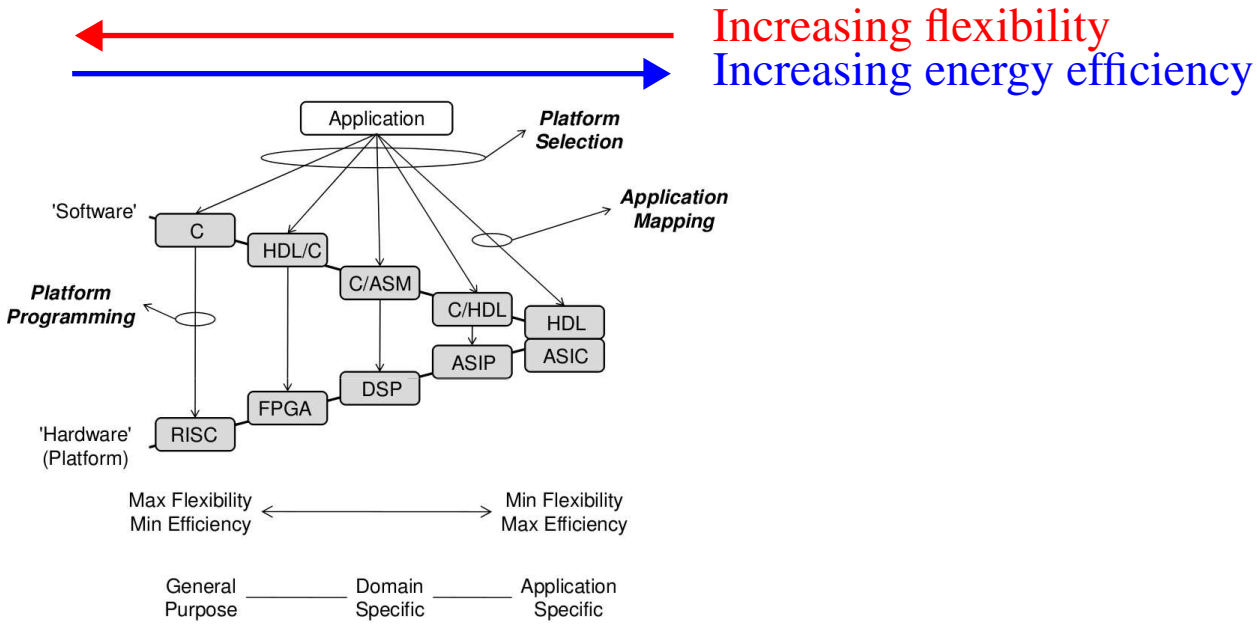


Fig. 1.7 The Hardware-Software Codesign Space

Flexibility refers to the versatility of the platform for implementing different application requirements, and how easy it is to update and fix bugs

Efficiency refers to performance (i.e. **time**-efficiency) or to energy efficiency

The Hardware-Software Codesign Space

Another concept reflected in the wedge-figure is the **domain-specific platform**

General-purpose platforms, such as RISC and FPGA, are able to support a broad range of applications

Application-specific platforms, such as the ASIC, are optimized to execute a single application

In the middle is a class called ***domain-specific*** platforms that are optimized to execute a **range of applications** in a particular application domain

Signal-processing, cryptography, networking, are examples of domains

And domains can have *sub-domains*, e.g., voice-signal processing vs. video-signal processing

Optimized platforms can be designed for each of these cases

DSPs and ASIPs are two examples of domain-specific platforms

The Hardware-Software Codesign Space

Codesign involves the following three activities:

- Platform selection
- Application mapping
- Platform programming

We start with a **specification**:

For example, a new application can be a novel way of encoding audio in a more economical format than current encoding methods

Designers can optionally write C programs to implement a prototype

Very often, a specification is just a piece of English text, that leaves many details of the application undefined

Step 1: **Select a target platform**

This involves choosing one or more programmable component as discussed previously, e.g., a RISC micro, an FPGA, etc.

The Hardware-Software Codesign Space

Step 2: **Application mapping**

The process of mapping an application onto a platform involves writing C code and/ or VHDL/verilog

Examples include:

- **RISC:** Software is written in C while the hardware is a processor
- **FPGAs**: Software is written in a hardware description language (HDL)
  FPGAs can be configured to implement a soft processor, in which case, software also needs to be written in C
- **DSP:** A digital signal processor is programmed using a combination of C and assembly, which is run on a specialized processor architecture
- **ASIP:** Programming an ASIP is a combination of C and an HDL description
- **ASIC**: The application is written in a HDL which is then synthesized to a hardwired netlist and implementation
  Note: ASICs are typically non-programmable, i.e., the application and platform are one and the same

The Hardware-Software Codesign Space

Step 3: **Platform programming** is the task of mapping SW onto HW

This can be done automatically, e.g., using a C compiler or an HDL synthesis tool

However, many platforms are **not** just composed of simple components, but rather require multiple pieces of software, possibly in different programming languages

For example, the platform may consist of a RISC processor and a specialized hardware coprocessor

Here, the software consists of C (for the RISC) as well as dedicated coprocessor instruction-sequences (for the coprocessor).

Therefore, the reality of platform programming is more complicated, and automated tools and compilers are NOT always available

The Hardware-Software Codesign Space

Difficult questions:

- How does one select a platform for a given specification (harder problem of two)
- How can one map an application onto a selected platform

The first question is harder - seasoned designers choose based on their previous experience with similar applications

The second issue is also challenging, but can be addressed in a more systematic fashion using a **design methodology**

A **design method** is a systematic sequence of steps to convert a specification into an implementation

Design methods cover many aspects of application mapping

- Optimization of memory usage
- Design performance
- Resource usage
- Precision and resolution of data types, etc.

A **design method** is a canned sequence of design steps

The Dualism of Hardware and Software

The **modeling** and **design** processes are very different between hardware and software, even using the simple single-clock synch. and single thread sequential models

In fact, HW and SW are the dual of each other in several respects:

- **Design Paradigm:** Parallel vs. sequential operation

  Hardware supports *parallel execution* of operations, while software supports *sequential execution* of operations

  The natural parallelism available in hardware enables more work to be accomplished by **adding more elements**

  In contrast, adding more operations in software increases its execution time

  Designing requires the decomposition of a specification into low level primitives such as gates (HW) and instructions (SW)

  Hardware designers develop solutions using **spatial decomposition** while software designer use **temporal decomposition**

The Dualism of Hardware and Software

- **Resource Cost**: Temporal vs. spatial decomposition

  The resources of hardware and software are *duals*

  When more resources are allocated in hardware, circuit complexity and area increase

  When more software operations are added, execution time increases

  Therefore, resource cost for hardware is circuit area while resource cost for software is execution time

- **Flexibility**

  Flexibility is the *ease* in which an application can be modified or adapted

  Software clearly excels over hardware with regard to flexibility

  - Flexibility is easily implemented in software and is essentially 'free'
  - In hardware, flexibility is *non-trivial*. It requires the **reuse** of circuit elements for different activities/functions in a design

The Dualism of Hardware and Software

- **Parallelism**

A dual of flexibility is parallelism, i.e., the ease in which parallel implementations can be created

For hardware, parallelism comes *for free* as part of the design paradigm

For software, parallelism is a major challenge

For single processor systems, software can only implement ***concurrency*** using special programming constructs called threads

For multi-processor systems, true parallelism can be realized but is complicated by inter-processor communication and synchronization

The Dualism of Hardware and Software

- **Modeling**

In software, modeling and implementation are similar

A C program is the *model*, its compilation is its *implementation*

Compilation produces assembly and then machine code but often the only representation that software engineers interact with is the programming language

In hardware, models and implementations of a design are **distinct**

A circuit is first described (modeled) using HDL or as a schematic

This representation can be simulated but it is not an implementation of the actual circuit

In order to implement it, *hardware synthesis* is required

Synthesis transforms the HDL representation to logic gates and then possibly to transistors and wires

The Dualism of Hardware and Software

- **Reuse**

  HW and SW are also different with regard to intellectual property reuse (IP-reuse)

  IP-reuse involves designing a component of a larger circuit or program such that it can be used in different designs

  In software, IP-reuse has proliferated through *open source*

  Today, designers start with a set of standard libraries that are well documented and implemented on a wide variety of platforms

  For hardware, IP-reuse is still maturing

  Today, designers are beginning to define standard exchange mechanisms, e.g., Spirit and Open EDA

In order to work effectively in codesign, you need to develop skills that allow you to translate easily between hardware and software, while considering the nature of this dualist relationship

Abstraction

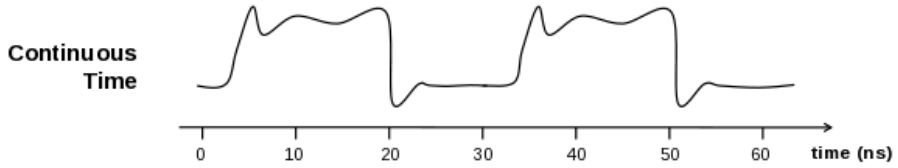
Abstraction refers to the ***level of detail*** that is available in a model

Lower levels have more detail, but are often much more complex and difficult to manage

Abstraction is heavily used to design hardware systems, and the representations at different levels are very different

A concept of abstraction is well exemplified by **time-granularity** in simulations

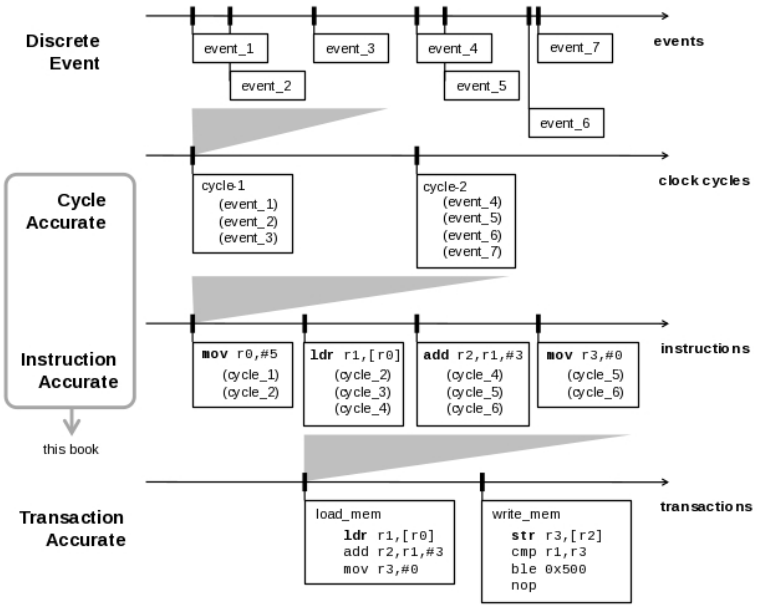
• **Continuous time** (lowest level):



Here, operations are described as continuous actions, e.g., using differential equations that model the charging and discharging of circuit nodes

This low level of modeling provides lots of details about circuit behavior, but is too compute-intensive and slow for codesign

Abstraction



- **Discrete-event**:

  Here, simulators abstract node behavior into discrete, possibly irregularly spaced, time steps called *events*

  Events represent the changes that occur to circuit nodes when the inputs are changed in the test bench

  The simulator is capable of modeling *actual propagation delay* of the gates, similar to what would happen in a hardware instance of the circuit

Abstraction

- **Discrete-event**: (cont.)

  Discrete-event simulation is very popular for modeling hardware at the lowest layer of abstraction in codesign

  This level of abstraction is much less compute-intensive than continuous time but accurate enough to capture details of circuit behavior including glitches

- **Cycle-accurate:**

  In this case, simulators model events only at regularly-spaced intervals, i.e., the rising edge of the clk (we talked about this earlier)

  A cycle-accurate model **does not** model propagation delays and glitches, i.e., all signals propagate with zero delay in combinational circuits

  This level of abstraction is considered the ***golden reference*** in HW/SW code-sign

Abstraction

- **Instruction-accurate:**

  Simulation of RTL models may be *too slow* for complex systems, e.g., your laptop has a processor that probably clocks over 1 GHz (one billion cycles/second)

  Instruction-accurate modeling expresses activities in steps of **one microprocessor instruction** (not cycle count)

  If you need to determine the real-time performance of a model, you need to translate instruction count to clk cycle count to obtain execution time

  Instruction-accurate simulators are used extensively to verify complex **software** systems

- **Transaction-accurate:**

  For very complex systems, even instruction-accurate models may be too slow or require too much modeling effort

  In transaction-accurate modeling, only the *interactions* (transactions) that occur between components of a system are of interest

Abstraction

- **Transaction-accurate (cont.):**

  For example, suppose you want to model a system in which a user process is performing hard disk operations, e.g., writing a file

  The simulator simulates commands exchanged between the disk drive and the user application

  The sequence of instruction-level operations between two transactions can number in the millions but the simulator instead simulates a single function call

  Transaction-accurate models are important in the **exploratory phases** of a design, before effort is spent on developing detailed models

For this course, we are interested in **instruction-accurate** and **cycle-accurate** levels

Concurrency and Parallelism

Both *concurrency* and *parallelism* occur often in HW/SW codesign, and they mean very different things

- Concurrency refers to *simultaneous execution* where the individual operations are completely **independent**
- Parallelism, on the other hand, refers to *simultaneous execution* where the operations are run on different processors or circuit elements

Therefore, concurrency refers to an **application model** while parallelism refers to the **implementation** of that model

Hardware is always *parallel*

Software can be *sequential*, *concurrent* or *parallel*

Sequential or concurrent software require only a single processor, while parallel software requires multiple processors

Software running on your laptop, e.g., WORD, email, etc. is concurrent

Software running on a 65536-processor IBM Blue Gene/L is parallel

Concurrency and Parallelism

A key objective of HW/SW codesign is to allow designers to leverage the benefits of true parallelism in cases where concurrency exists in the application

There is a well-known Comp. Arch principle called **Amdahl's law**

The maximum speedup of any application that contains q% sequential code is:

$$\frac{1}{\left(\frac{q}{100}\right)}$$

For example, if your application spends 33% of its time running sequentially, the maximum speedup is 3

This means that no matter how fast you make the parallel component run, the maximum speedup you will ever be able to achieve is 3

The task of making your application take advantage of parallelism is **not obvious**

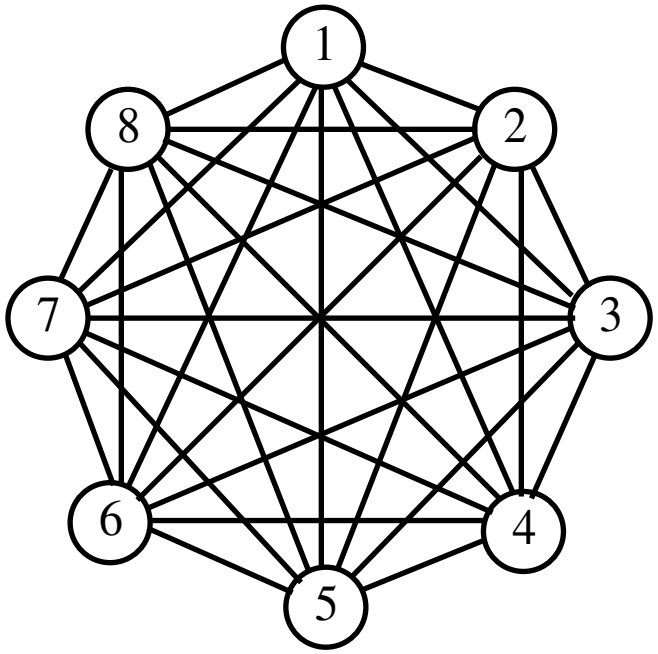
C programs are sequential, and so are typical instruction set architectures

Concurrency and Parallelism

Consider an application that performs addition, and assume it is implemented on a Connection Machine (from the'80s)

The **Connection Machine** (CM) is a massively parallel processor, with a network of processors, each with its own local memory

Connection Machine:
Original machine contained 65556 processors, each with 4Kbits of local memory



Completely connected

How hard is it to write programs for this machine?

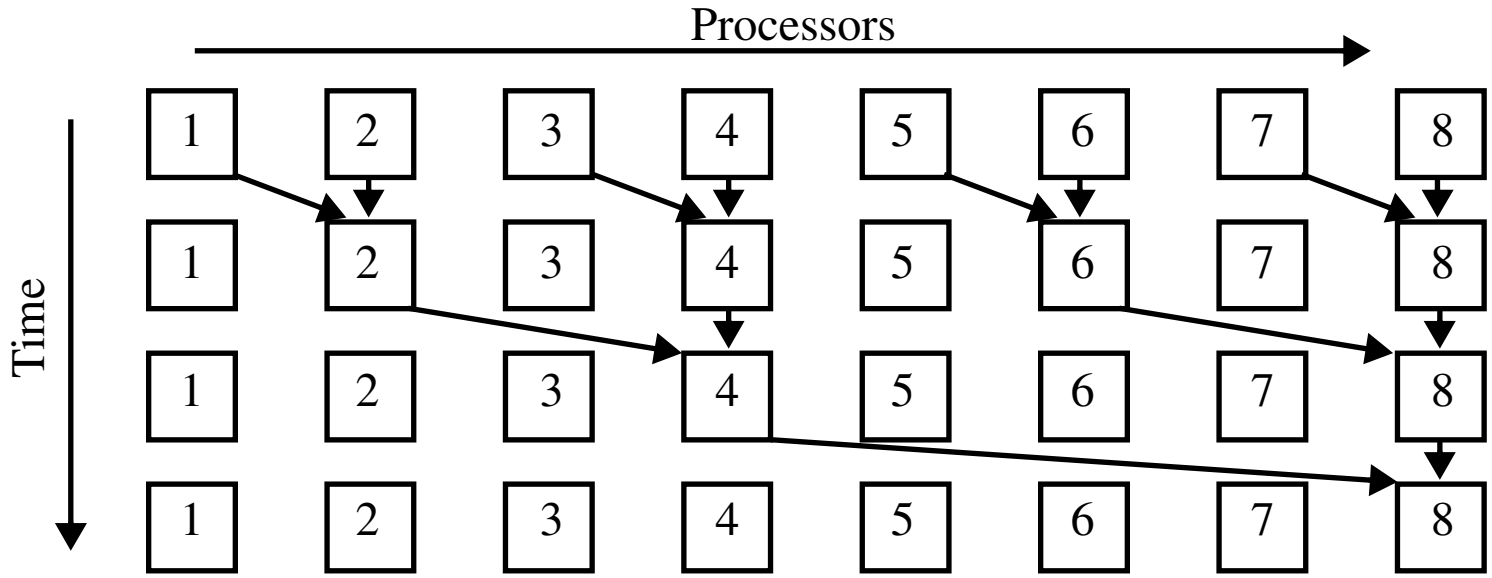
It's possible to write individual C programs for each node, but this is really not practical with 64K nodes!

Concurrency and Parallelism

The authors of the CM, Hellis and Steele, show that it is possible to express algorithms in a **concurrent** fashion so that they map neatly onto a CM

Consider the problem of summing an array of numbers

The array can be distributed across the CM by assigning one number to each processor



The sum is computed in *log(n)* steps, i.e., in only 3 time steps in this example

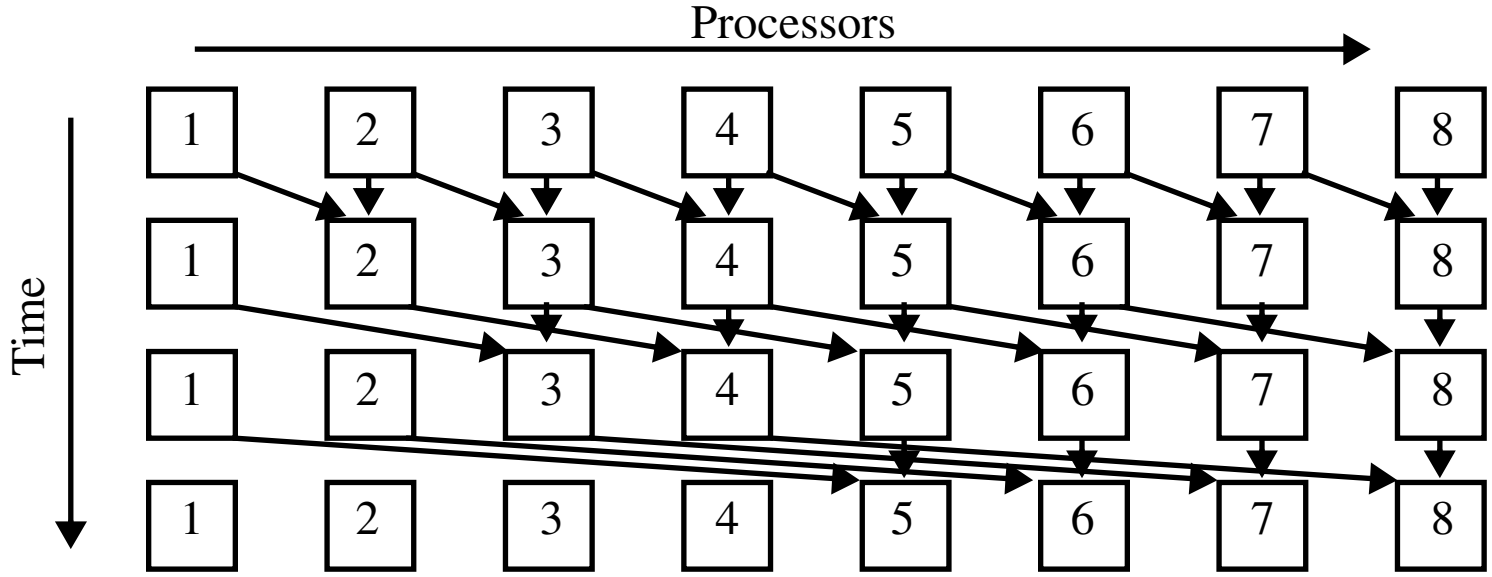
The same algorithm running on a sequential processor would take 7 steps

Concurrency and Parallelism

Even through the *parallel sum* speeds up the computation significantly, there remains a lot of wasted compute power

Compute power of a smaller 8-node CM for 3 times steps is 3*8 = 24 computation time-steps of which only 7 are being used

On the other hand, if the application requires *all partial sums*, i.e., the sum of the first two, three, four, etc. numbers, then the full power of the parallel machine is used



Here, 17 computation time-steps are used

Concurrency and Parallelism

There are many other data-parallel versions of algorithms that are intuitively sequential (see Hillis and Steele)

Key Take-Away: You can ONLY leverage the full power of the underlying parallelism in the hardware if you develop a *concurrent specification*

If you restrict yourself to a sequential specification, it will be much harder to leverage the underlying parallel hardware

You should **not** settle for sequential programming languages such as C when developing codesign solutions

There are existing concurrent specification mechanisms, such as **data-flow** (to be discussed), that are much better suited for parallel implementations

Data Flow Models

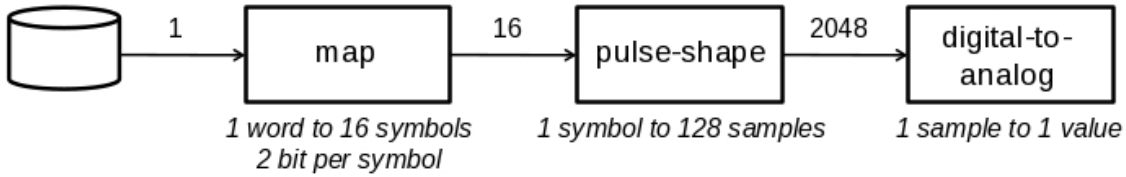
As we discussed, hardware is parallel while software is sequential

We need **concurrent** high-level models to allow designers to describe systems that are independent of software or hardware

This provides flexibility in mapping components to either HW or SW in later phases of design

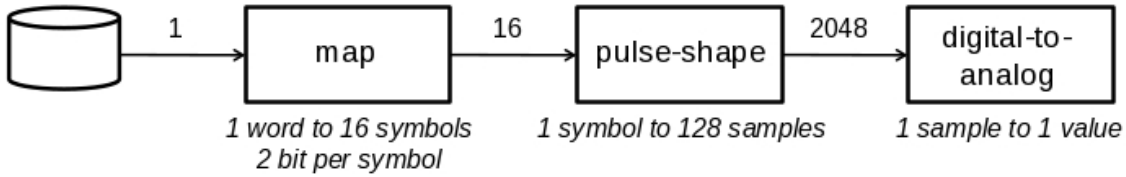
Block diagrams have become popular as a mechanism to describe systems, including DSP systems such as digital radios and radar, at a high level of abstraction

Block diagrams use symbols to represent signal processing functions, i.e., operations performed on a digital data stream, without specifying the implementation strategy



This block diagram shows an example of a ***pulse-amplitude modulation*** (**PAM**) system, used to transmit information over bandwidth-limited channels

Data Flow Models

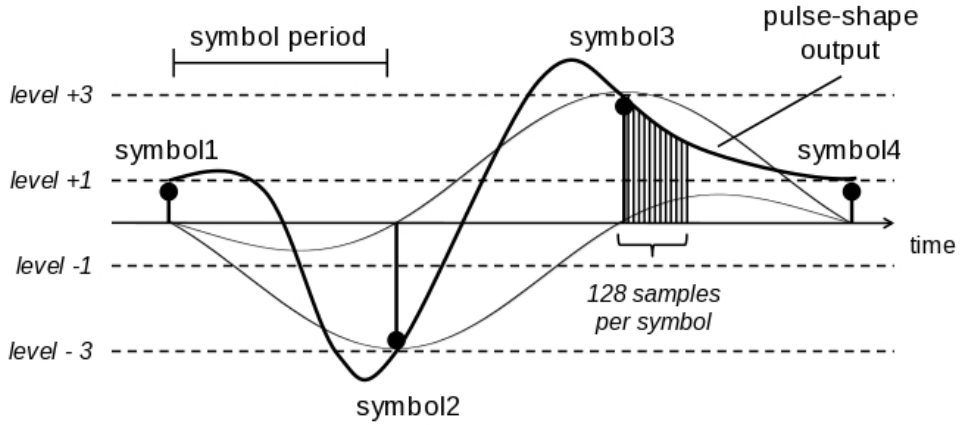


The PAM system reads a file of binary data in 32-bit chunks

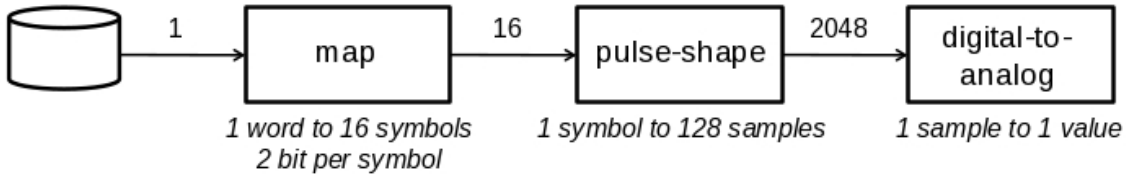
The *map* function converts the 32-bits of each word to 16 2-bit symbols (PAM-4 is used to refer to such systems

Each 2-bit sequence maps to one of 4 separate symbols in the set {-3, -1, 1 and 3}

Each symbol represents a *pulse height*



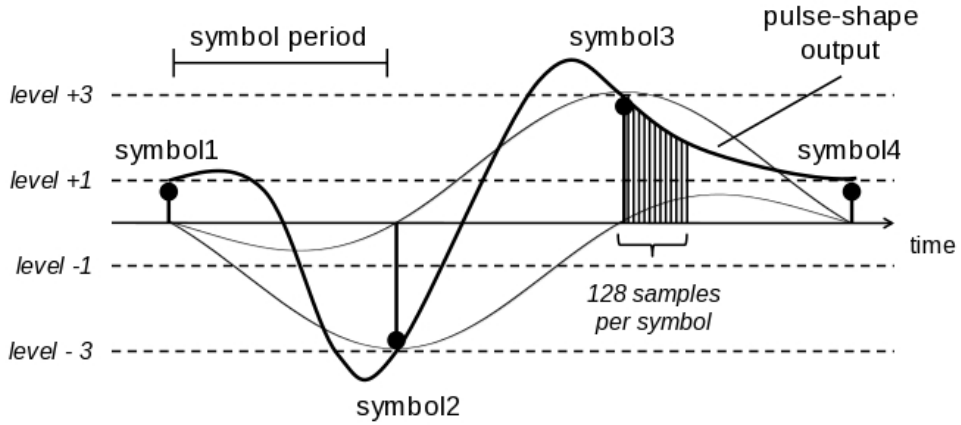
Data Flow Models



The 2-bit symbols need to be converted into a smooth shape using *pulse shaping*

The *pulse shaper* ensures that the frequency content of the smoothed curve does not exceed 2X the symbol rate to avoid adding artifacts to the generated curve

The *pulse-shape* function **oversamples** the symbols to provide a smooth function, producing 128 digital samples for each symbol as shown



A convolution-based function is used to ensure the curve goes through the symbols while providing a *memory* effect, which allows symbols to influence other symbols

Data Flow Models

The *digital-to-analog converter* (DAC) is used to convert the discrete samples into an analog output signal

Here's a high-level simulation model for PAM-4

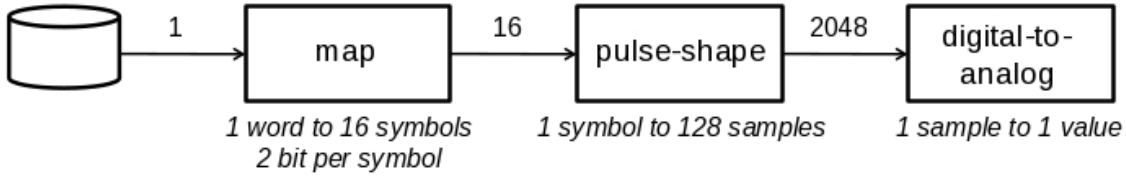
```
extern int read_from_file(), map_to_symbol(int, int);
extern int pulse_shape(int, int);
extern void send_to_da(int);
int main() {
   int word, symbol, sample;
   int i, j;
   while (1) {
      word = read_from_file();
      for ( i = 0; i < 16; i++ ) {
         symbol = map_to_symbol(word, i);
         for ( j = 0; j < 128; j++ ) {
            sample = pulse_shape(symbol, j);
            send_to_da(sample);
} } } }
```

Data Flow Models

Although this is a good model for simulation, it is **not** for an implementation

C implicitly assumes sequential execution

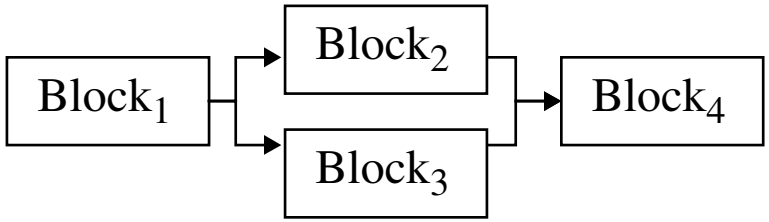
The block diagram, on the other hand, is implicitly parallel



The lines between blocks represent *data dependencies*, and therefore, they force an ordering to the sequence of operations

However, unlike C, each block can execute simultaneously with other blocks

Another example of the difference between C and block diagrams is shown by the 'fanout' in the following:



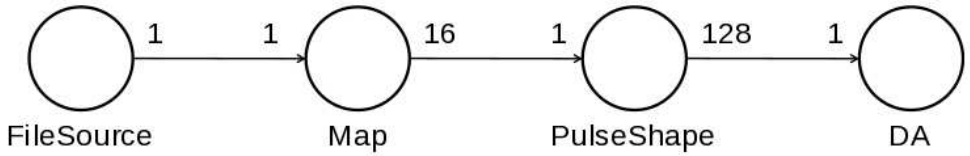
Here, $Block_2$ and $Block_3$ are clearly parallel but C would execute them sequentially

Data Flow Models

Note that, in general, it is *easier* to create a sequential implementation from a parallel model than it is to create a parallel implementation from a sequential model

This argues in favor of **Data Flow** diagrams for modeling

The following is a Data Flow model of PAM-4:



The bubbles, called **actors**, represent the functions in the block diagram

Actors are linked together using directional lines, called **queues**

The numbers on the lines represent the relative **rates** of communications between modules, e.g., *Map* converts a 32-bit word into 16 2-bit symbols

Note that each *actor* works **independently**, i.e., it checks its input queue for the proper number of elements and executes immediately when satisfied

Data Flow Models vs. C Programs

We cover Data Flow extensively in this lecture series but for now, we note the following important differences between C and Data Flow models:

- Data Flow is a ***concurrent model*** (this is a major driver for their popularity), which means they can easily be mapped to hardware or software implementations
- Data Flow models are *distributed*, i.e., there is no centralized controller, i.e., each *actor* operates autonomously
- Data Flow models are *modular*, allowing libraries of components to be constructed and utilized in a *plug-and-play* fashion
- Data Flow models can be *analyzed*, e.g., for *deadlock* conditions that can result in system lock-up

  Deterministic, mathematical methods can be used to analyze Data Flow models, which is generally not possible using C

Data Flow models have been around since the early 1960s

The 70's and 80's were active periods of research and development of Data Flow programming languages and even Data Flow architectures

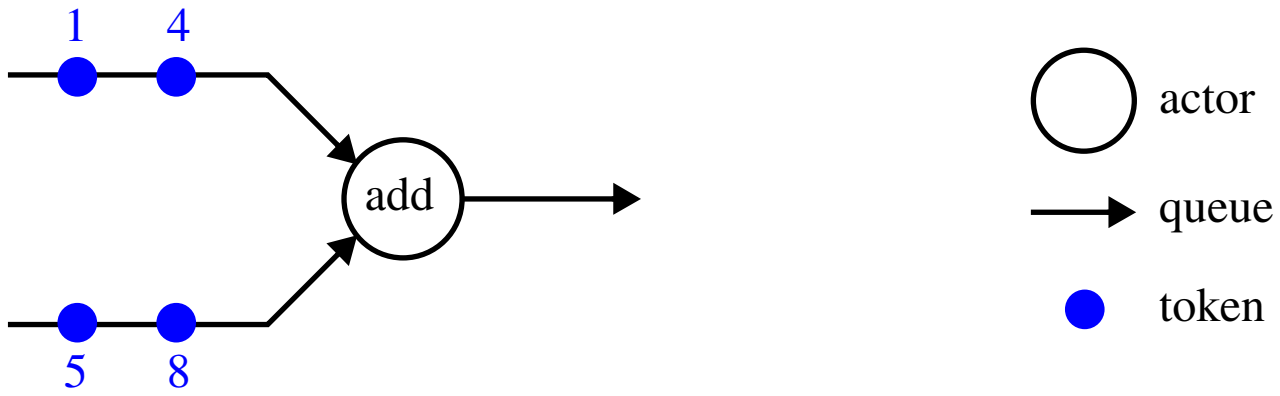
NI's **Labview** is a classic example of a Data Flow programming language

Tokens, Actors and Queues

Here, we define the elements that make up a Data Flow model, and discuss a special class of Data Flow models called *Synchronous Data Flow* (SDF) Graphs

SDFs allow for the application of formal analysis techniques

A simple example:



A Data Flow model is made up of three elements:

• **Actors**: Contain the actual operations

Actors have a precise beginning and end, i.e., they have *bounded* behavior, and they *iterate* that behavior continuously

Each iteration is called a ***firing***, e.g., an addition is performed on each firing

Tokens, Actors and Queues

- **Tokens**: Carry information from one *actor* to another

  A token has a *value*, such '1' and '4' as shown above

- **Queues**: Unidirectional communication links that transport tokens between *actors*

  We assume Data Flow *queues* have an **infinite** amount of storage

  Data Flow *queues* are first-in, first-out (FIFO)

  In above example, token '1' is entered after token '4' so token '4' is processed first

When a Data Flow model executes, *actors* read tokens from their **input** *queues*, apply an operation and then write values to the **output** *queue*

The execution of a Data Flow model is expressed as a sequence of **concurrent actor firings**

Tokens, Actors and Queues

Data Flow models are **untimed**

The firing of an *actor* happens instantaneously and therefore time is **irrelevant**

Firings actually take non-zero time in an actual implementation

The execution of Data Flow models is *guided* only by **the presence of data**, i.e., an *actor* can **not** fire until data becomes available on its inputs

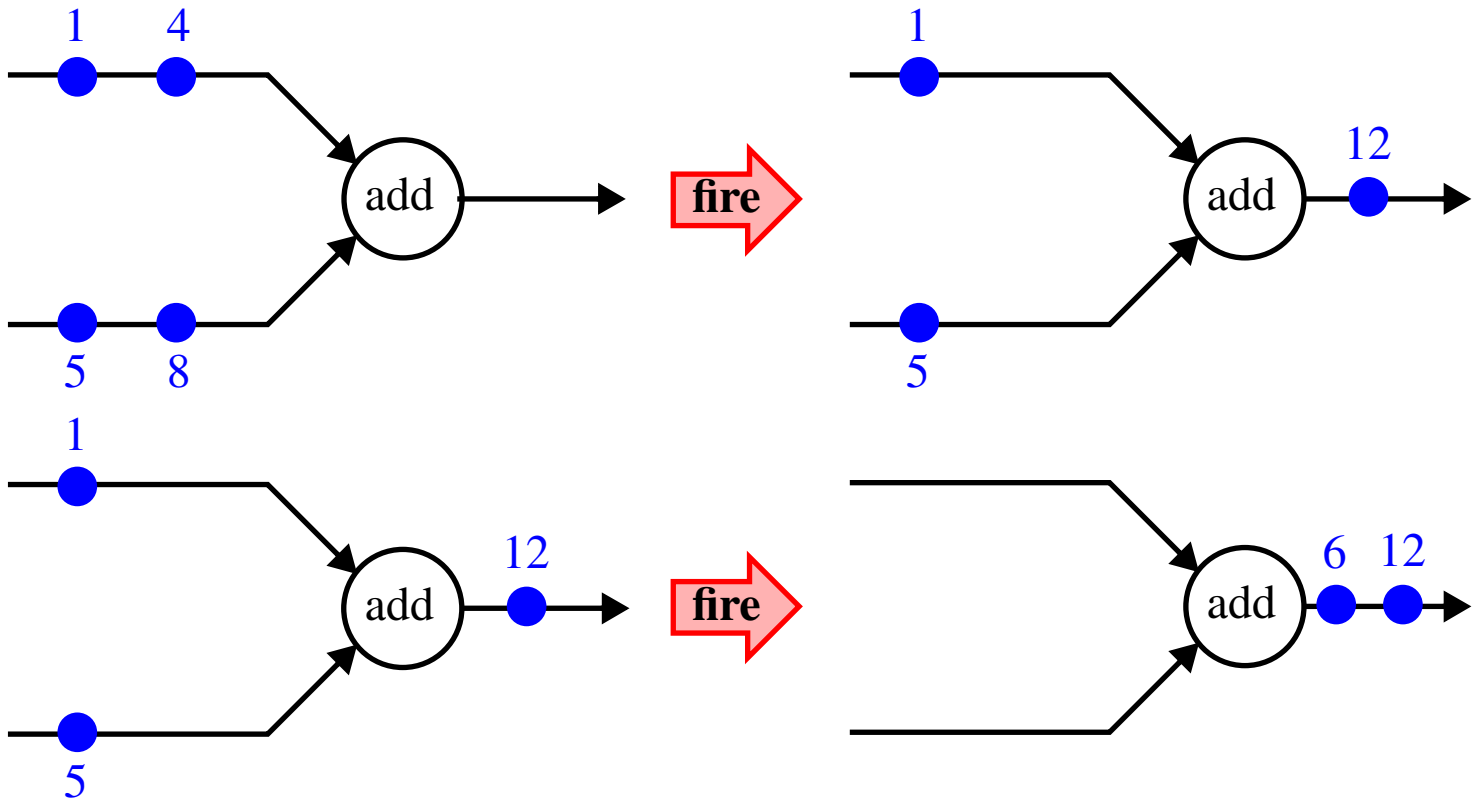
A Data Flow graph with tokens distributed across its *queues* is called a **marking** of a Data Flow model

A Data Flow graph goes through a series of *marking* when it is executed

Each marking corresponds to a different **state** of the system

The distribution of tokens in the *queues* (marking) are the ONLY observable state in the system (no state is maintained inside the actors)

Firing Rates, Firing Rules and Schedules



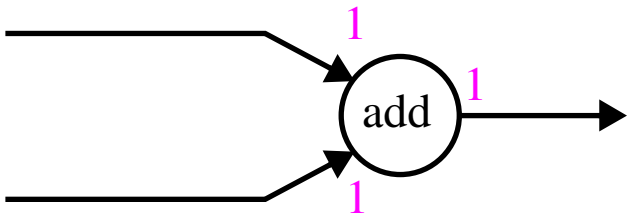
A **firing rule** defines the conditions that enable an *actor* to fire

In the above example, the firing rule checks that the *actor's* input *queues* contain at least one token

Therefore, *actors* are able to check the number of tokens in each of its *queues*

Firing Rates, Firing Rules and Schedules

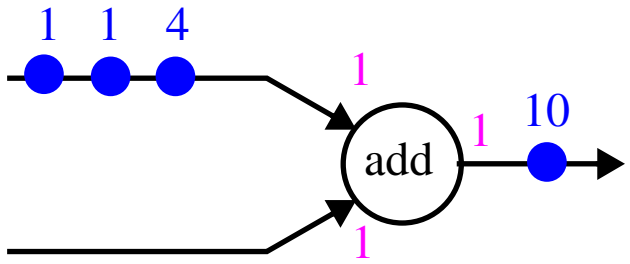
The **required** number of tokens consumed and produced can be annotated on the *actors* inputs and outputs, respectively



Inputs: **token consumption rate**

Outputs: **token production rate**

Therefore, this information combined with a marking makes is easy to decide whether an *actor* can fire



Not able to fire

Data Flow *actors* can also consume *more than one* token per firing

This is referred to as a **multi-rate** Data Flow graph

Synchronous Data Flow Graphs

Synchronous Data Flow (SDF) graphs refer to systems where the number of tokens consumed and produced per actor firing is ***fixed*** and ***constant***

The term *synchronous* refers to the **fixed** consumption and production rate of tokens

Note that SDF will **not** be able to handle *control-flow* constructs, such as *if-then-else* statements in C without adding special operators (which we will discuss)

Despite this significant limitation, SDFs are very powerful (and popular), and more importantly, mathematical techniques can be used to verify certain properties

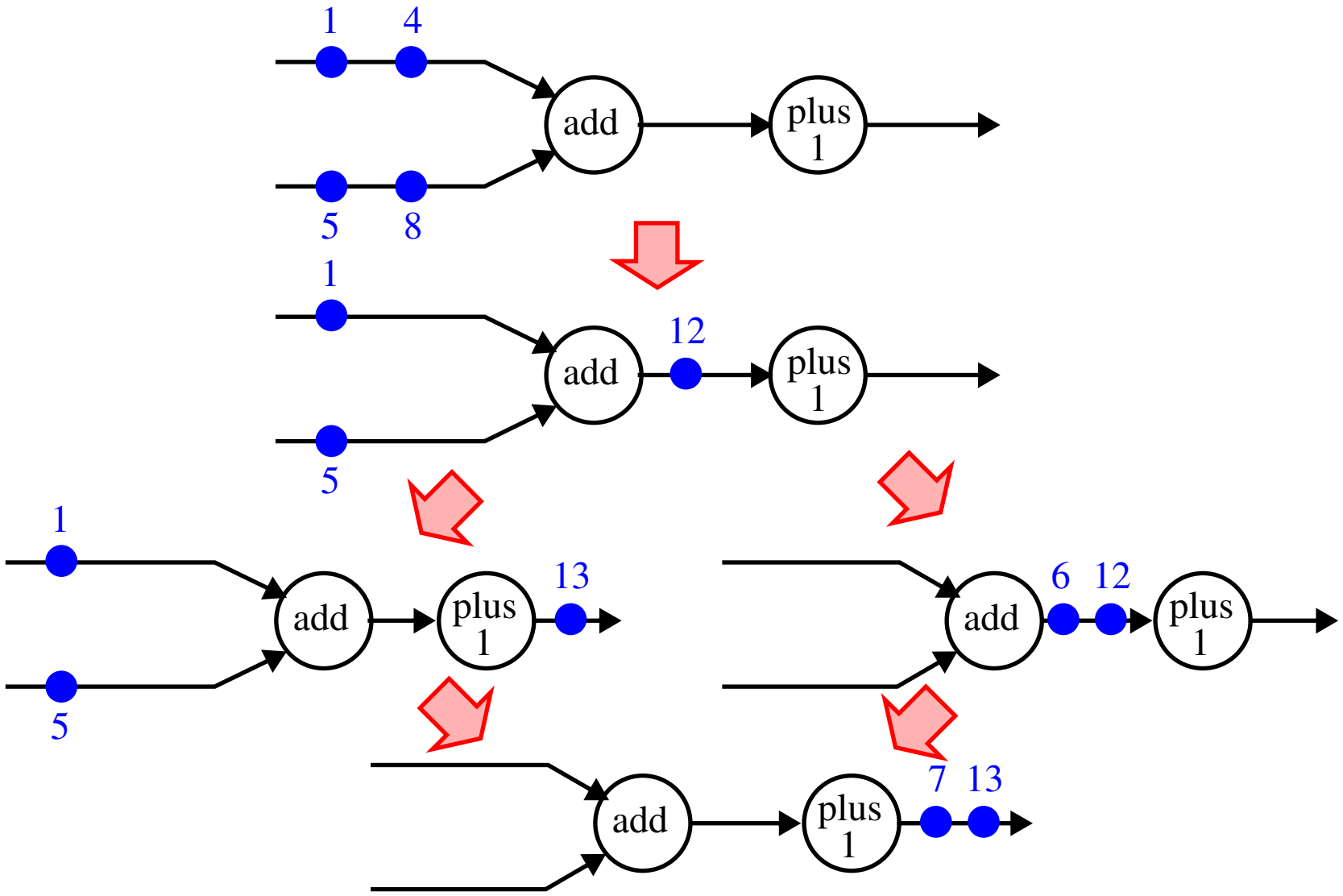
The first of these properties is **determinism**

The entire SDF is deterministic under the condition that all of its *actors* implement a deterministic function

Determinism guarantees that the same results will always be produced **independent of the firing order**

SDF Graphs

Illustration of determinism:



SDF Graphs

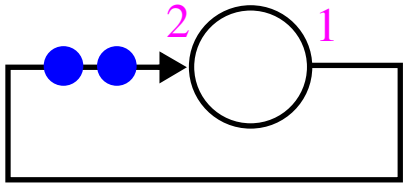
A significant benefit of determinism is that it allows arbitrary mappings of the *actors* onto parallel architectures while guaranteeing the same results

For example, correct results are obtained even if we execute the *add actor* on a fast processor and the *plus1 actor* on a slow processor

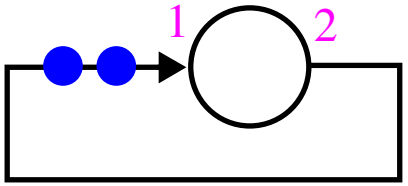
The second important property of SDF relates to an ***admissible schedule***

An **admissible** SDF is one that can run forever without *deadlock* (unbounded execution) or without overflowing any of the communication queues (bounded buffer)

Deadlock occurs when an SDF graph progresses to marking that prevents firings



Graph is deadlocked



Infinite # of tokens produced

Overflow occurs when tokens are produced faster than they are consumed

SDF Graphs

There is also a systematic method to determine whether a SDF graph is *admissible*

The method provides a closed form solution, i.e., no simulation is required

Lee proposed a method called **Periodic Admissible Schedules** (PASS), defined as:

- A *schedule* is the order in which the actors must fire
- An *admissible schedule* is a firing order that is deadlock-free with bounded buffers
- A *periodic admissible schedule* is a schedule that supports *unbounded execution*, i.e., is periodic in the sense that the same markings will recur

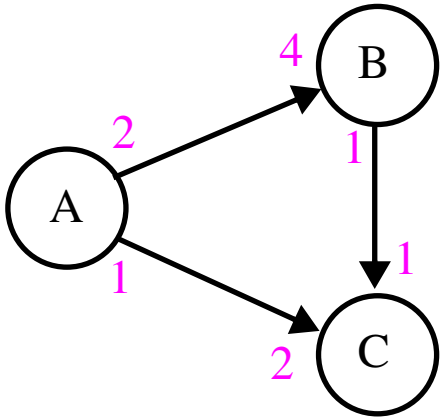
We also consider a special case called *Periodic Admissible* ***Sequential*** *Schedules* (PASSs) that supports a microprocessor implementation with one *actor* firing at a time

There are four steps to creating a PASS for an SDF graph:

- Create the topology matrix **G** of the SDF graph
- Verify the *rank* of the matrix to be one less than the number of nodes in the graph
- Determine a firing vector
- Try firing each actor in a *round robin* fashion, until the *firing count* given by the firing vector is reached

SDF Graphs

Consider the following example:



Step 1: Create a topology matrix for this graph:

The topology matrix has as many rows as there are *edges* (FIFO queues) and as many columns as there are *nodes*

The entry (*i, j*) will be positive if the node *j* **produces** tokens onto the edge *i* and negative if it consumes tokens

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{array}{l} \leftarrow edge(A,B) \\ \leftarrow edge(A,C) \\ \leftarrow edge(B,C) \end{array}$$

NOTE: This matrix do NOT need to be square

SDF Graphs

Step 2: The condition for a PASS to exist is that the *rank* of **G** has to be one less than the number of nodes in the graph

The *rank* of the matrix is the number of **independent equations** in **G**

For our graph, the rank is 2 -- verify by multiplying the first column by -2 and the second column by -1, and adding them to produce the third column

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \Rightarrow G = \begin{bmatrix} -4 & +4 & 0 \\ -2 & 0 & -2 \\ 0 & -1 & -1 \end{bmatrix}$$

Given that there are *three* nodes in the graph and the rank of the matrix is 2, a PASS is **possible**

This step effectively verifies that tokens can **NOT** accumulate on any edge of the graph

The actual number of tokens can be determined by choosing a firing vector and carrying out a matrix multiplication

SDF Graphs

For example, the tokens produced/consumed by firing A twice and B and C zero times is given by:

$$\text{firing vector } q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \Rightarrow Gq = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix}$$

This vector produces 4 tokens on edge(A,B) and 2 tokens on edge(A,C)

Step 3: Determine a periodic firing vector

The *firing vector* given above is not a good choice to obtain a PASS because it leaves tokens in the system

We are instead interested in a firing vector that leaves no tokens:

$$Gq_{\text{PASS}} = 0$$

Note that since the *rank* is less than the number of nodes, there are an infinite number of solutions to the matrix equation

SDF Graphs

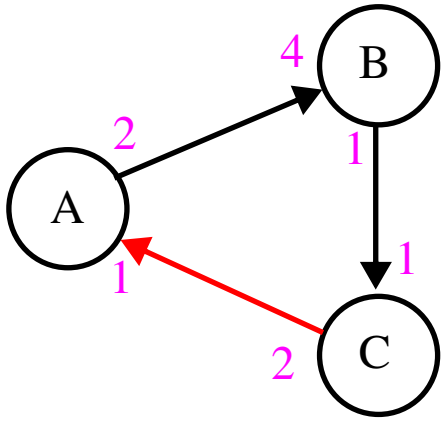
Step 3: Determine a periodic firing vector (cont.)

This is true b/c, intuitively, if *firing vector* (*a, b, c*) is a PASS, then so should be firing vectors (*2a, 2b, 2c*), (*3a, 3b, 3c*), etc.

Our task is to find the simplest one -- for this example, it is:

$$q_{\text{PASS}} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \Rightarrow Gq_{\text{PASS}} = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that the existence of a PASS firing vector does **not** guarantee that a PASS will also exist



Here, we reversed the (A,C) edge

We would find the same q_{PASS} but the resulting graph is **deadlocked** -- all nodes are waiting for each other

SDF Graphs

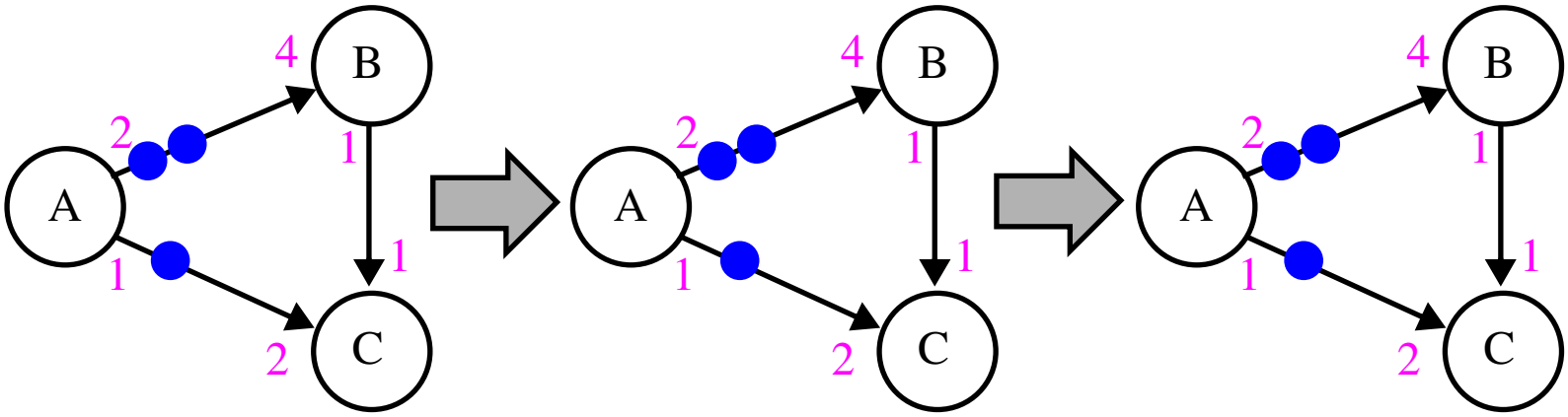
Step 4: Construct a *valid* PASS.

Here, we fire each node up to the number of times specified in q_{PASS}

Each node that is able to fire, i.e., has an adequate number of tokens, will fire

If we find that we can fire NO more nodes, and the firing count is **less** than the number in q_{PASS}, the resulting graph is **deadlocked**

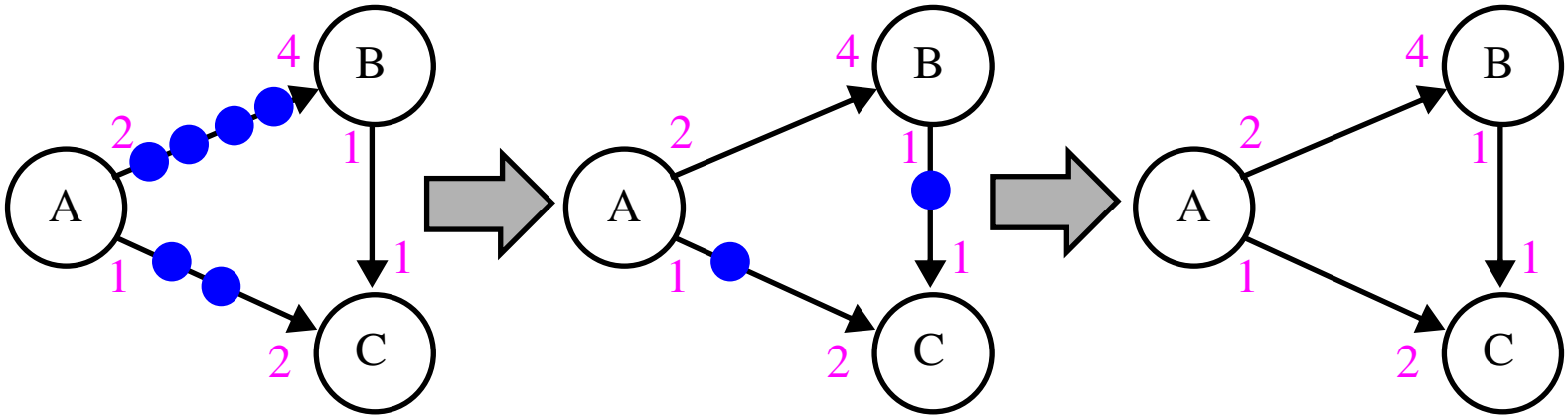
Trying this out on our graph, we fire *A* once, and then B and C



Fire *A* (succeeds) Fire *B* (FAILS -- not enough tokens) Fire *C* (FAILS)

SDF Graphs

Step 4: Construct a *valid* PASS.



Fire *A* AGAIN (succeeds) Fire *B* (succeeds) Fire *C* (succeeds)

So the PASS is (*A, A, B, C*)

Try this out on the **deadlocked** graph -- it aborts immediately on the first iteration because no node is able to fire successfully

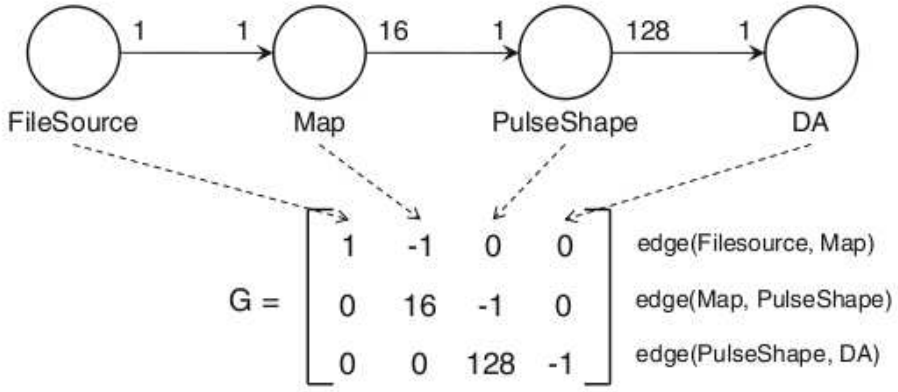
Note that the **determinate** property allows any ordering to be tried freely, e.g., *B, C* and then *A*

In some graphs (not ours), this may lead to additional PASS solutions

SDF Graphs: PAM-4 Example

Consider the *digital pulse-amplitude modulation system* (PAM-4) discussed earlier

The SDF for this system consists of 4 *actors*, and is a *multi-rate* Data Flow system:



The first step is to construct the *topology matrix G*

The *queues* correspond to the 3 rows and *actors* to the 4 columns

The second step is to verify the rank is the number of *actors* minus 1

It is easy to show that the 3 rows are independent, i.e., are not linear combinations of any other rows

This confirms that a PASS is possible

SDF Graphs: PAM-4 Example

The third step is to derive a feasible firing for the system

The firing vector, q_{PASS}, must yield a zero-vector when multiplied by the topology matrix

$$q_{\text{PASS}} = \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2048 \end{bmatrix} \Rightarrow Gq_{\text{PASS}} = \begin{bmatrix} +1 & -1 & 0 & 0 \\ 0 & +16 & -1 & 0 \\ 0 & 0 & +128 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2048 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The fourth step is to derive a schedule -- there are two possibilities

- The first one is trivial, fire each *actor* in succession, from left to right

  Note that the *queue* (FIFO) sizes are 16 and 2048

- Alternatively, we can fire *FileSource* and *Map* once and then repeat the following sequence: Fire *PulseShape* once and then fire *DA* 128 times

  The benefit here is the reduced *queue* sizes, i.e., the *PulseShape* input *queue* reduces from 16 to 1 while the *DA* input *queue* reduces from 2048 to 128

In general, deriving the optimal schedule is a difficult problem for complex systems

Limits of SDF Models

In conclusion, SDFs are very useful

They allow a designer to determine certain important system properties, such as the *determinism*, *deadlock*, and *storage requirements*

Unfortunately, SDFs are **not** a universal specification mechanism, in particular, SDFs do not have constructs to allow **control-flow** modeling

Control appears in different forms in system design:

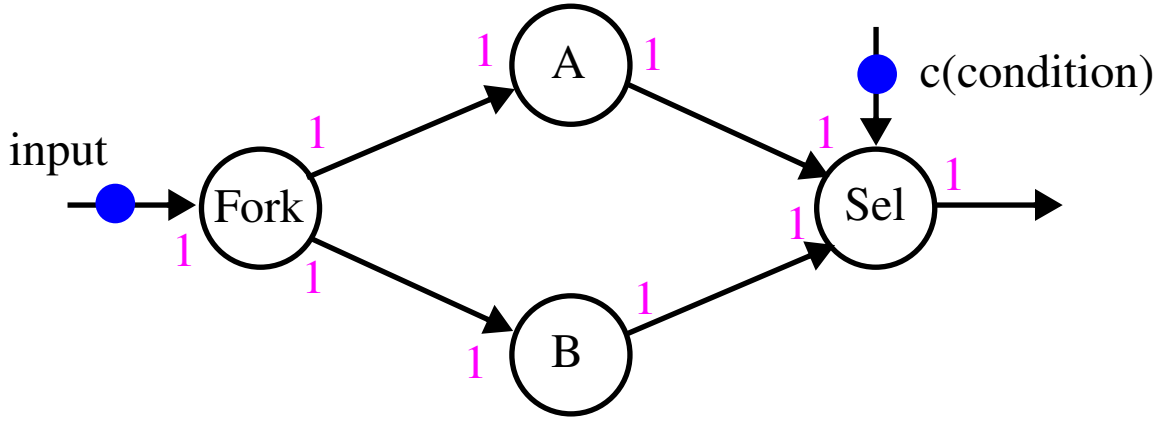
- Stopping and re-starting: An SDF model runs forever
  Stopping/re-starting is a control-flow property not addressed with SDFs
- Mode-switching: When a cell-phone switches from one standard to the other, the baseband processing (modeled as an SDF) needs to be reconfigured
  The topology of an SDF graph is fixed and **cannot** be modified at runtime
- Exceptions: Error conditions arrise in applications
  SDFs cannot model exceptions that affect the entire graph, e.g., empty queues
- Run-time conditions: A simple *if-then-else* stmt cannot be modeled by SDFs
  An SDF node does not support conditional execution -- it is always active

Limits of SDF Models

There are two solutions to the problem of *control flow modeling* in SDFs

Solution 1: *emulate* control flow on top of the SDF semantics

Consider the stmt *if (c) then A else B*



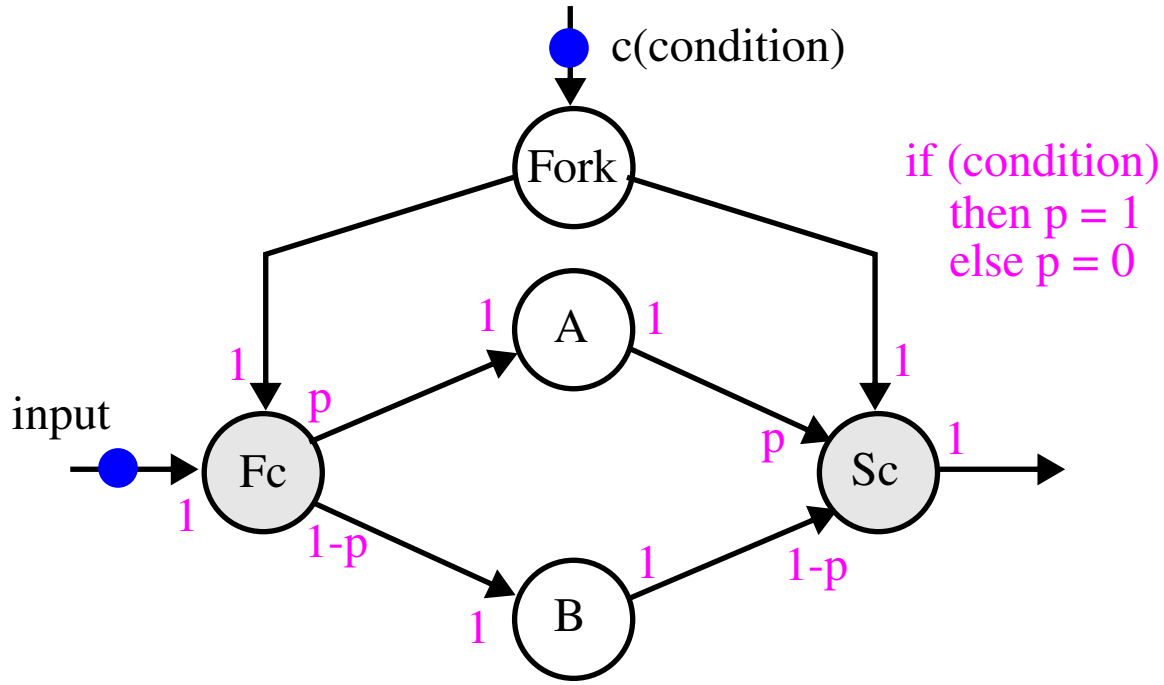
The *selector-actor* on the right routes either *A* or *B* to the output

Note that this is not an exact match to the *if-then-else* in C because BOTH the *if* branch (*A*) and the *else* (*B*) must execute and produce tokens

However, it is a good match to hardware, which uses a *multiplexer* to select among one of several input results

Limits of SDF Models

Solution 2: *extend* the SDF semantics using *Boolean Data Flow* (BDF)



BDFs 'tune' the *production* and *consumption* rate of a *actor* according to the value of an **external control token**

The *condition* token is distributed to two BDF **conditional fork** and **merge** nodes, *Fc* and *Sc*

Limits of SDF Models

Here, the *conditional fork* will fire when there is an *input* token AND a *condition* token

A token is produced on EITHER the upper or lower edge, dependent on the *condition* token

This is indicated by a dynamic variable p, which signifies a *conditional production rate*

The *conditional merge* works similarly, i.e., it fires when there is a *condition* token and will consume a token on EITHER the upper or lower edge

Unfortunately, BDFs detract from the usefulness of SDFs

For example, we now have Data Flow graphs that are *conditionally admissible*

Also, the topology matrix now includes symbolic values, p, which complicates the closed form math

For a SDF with 5 conditions, we would have a matrix with 5 symbols or would need to expand the single matrix into 32 variants (2^5)

Limits of SDF Models

Beyond BDF, other flavors of *control-oriented* Data Flow graphs have been proposed that have similar challenges, such as:

- Dynamic Data Flow (DDF) which allows variable production and consumption rates
- Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on production and consumption rates

DFG Performance Modeling and Transformations

We indicated earlier that Data Flow graphs (DFGs) are **untimed**, i.e., our analysis did not model the amount of time needed to complete a computation

In this lecture, we describe how to use DFGs for performance analysis

Performance estimation will be accomplished by modeling only two components: *actors* and *queues*

Once our new modeling constructs are introduced, we then turn our attention to *transformations* designed to enhance performance

Input sample rate is the time interval between two adjacent input samples from a data stream

For example, a digital sound system generates 44,100 samples per second

Input sample rate defines a **design constraint** for the *real-time* performance of the Data Flow system

Similar constraints usually exists for *output sample rate*

Definitions

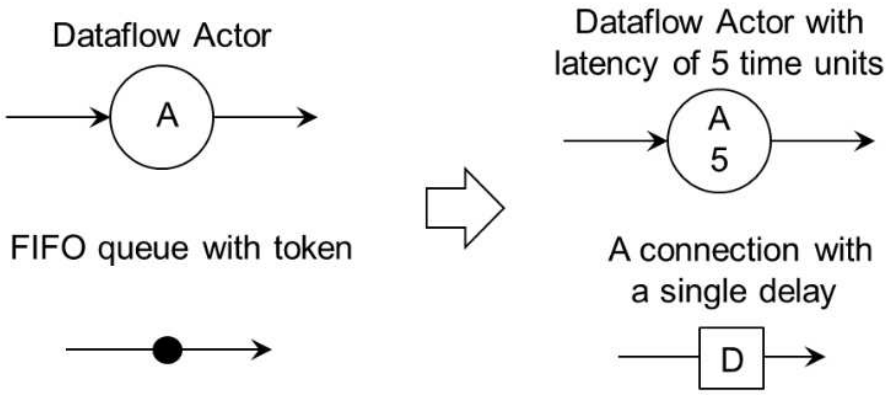
We use two common metrics as measures of performance:

- **Throughput**: the number of *samples* processed per second

  Note that input and output throughput may be different

- **Latency**: The time required to process a single token from input to output

The Data Flow Resource Model:



We used the symbols on the left earlier to model DFGs

For performance modeling, we

- Include a number within the *actor* symbol to model execution latency
- Replace FIFO *queues* with a communication channel, which includes delays

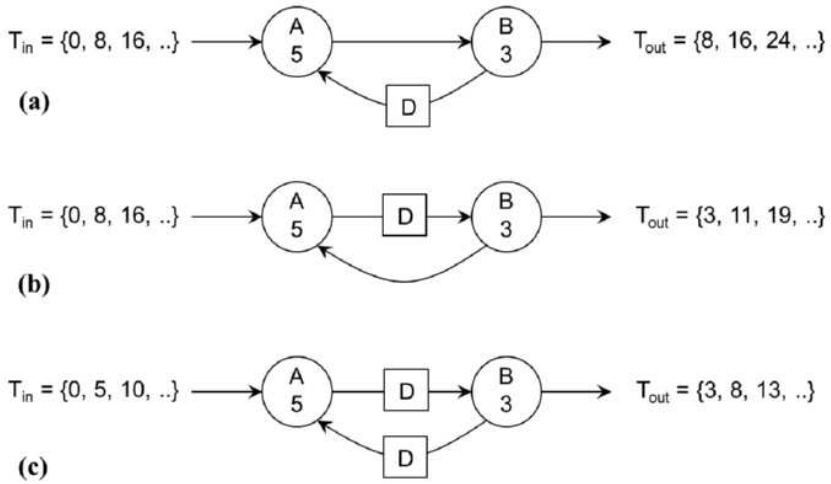
Definitions

Note that the number included in an *actor* represents the amount of time it takes (in clock cycles, nanoseconds, etc) **after** it fires

Time spent while waiting for input data is not counted

Also note that the *delay* element (which replaces FIFO *queues*) can hold exactly one token

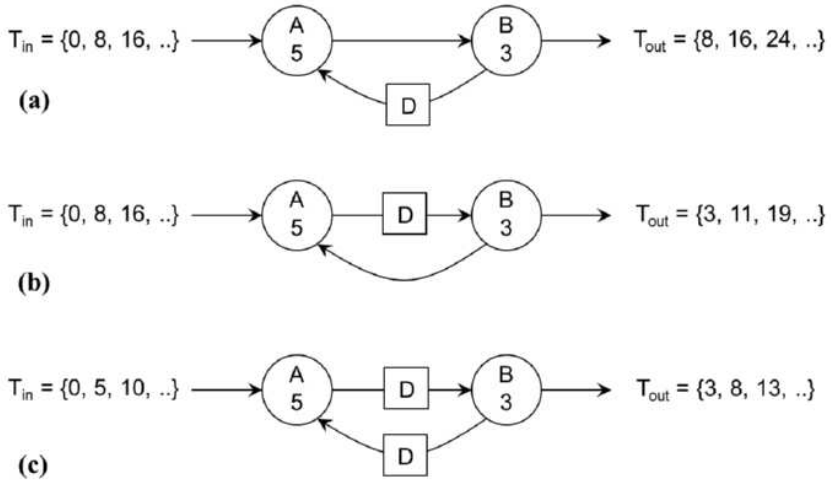
Think of *delay* elements as buffers with 1 unit of delay



We can use a performance annotated DFG to evaluate its execution time

In (a), (b) and (c) above, *actor* A introduces 5 units of latency while B introduces 3 units

Performance Analysis



The time stamp sequences on the left and right indicate when input samples are read and when output samples are produced

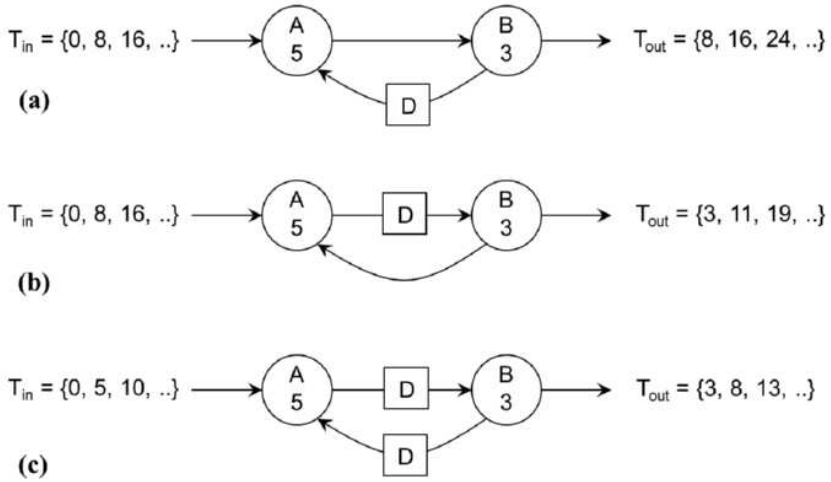
The time stamps for DFG (a) and (b) are different because of the position of the *delay* element in the loop

(a) requires the sum of execution times of A and B before producing a result

(b) can produce a result at time stamp 3 because the *delay* element allows it to execute immediately at system start time (we refer to this as *transient* behavior)

In this case, the *delay* elements affect only the latency of the first sample

Performance Analysis



In contrast, (c) shows that *delay* elements can be positioned to enable parallelism, and affect both latency and throughput

Both *actors* can execute in parallel in (c), resulting in better performance than (a) and (b)

The throughput of (a) and (b) is 1 sample per 8 time units, while (c) is 1 sample per 5 time units

Similar to a pipelined system, the throughput in (c) is ultimately **limited** to the speed of the **slowest** *actor* (A in this case -- B is forced to wait)

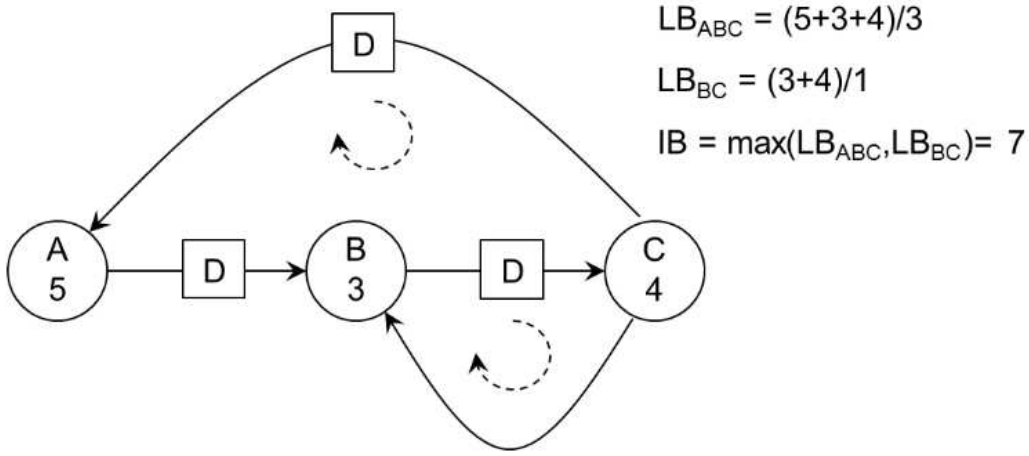
Limits on Throughput

As indicated, the distribution of the *delay* elements in the loops impacts performance

As an aid in analyzing performance, let's define

- ***Loop bound*** as the round-trip delay of a loop, divided by the number of delays in the loop
- ***Iteration bound*** as the *largest* loop bound in any loop of a DFG

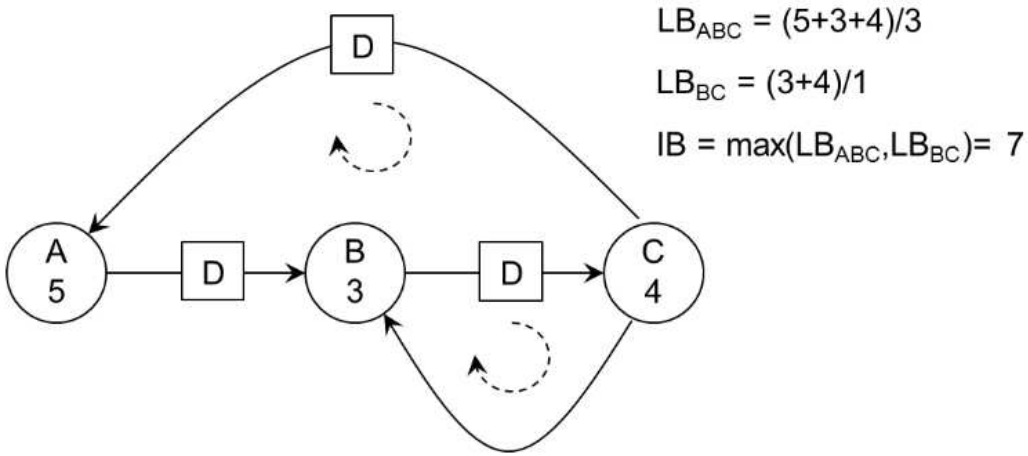
Iteration bound defines an upper limit on the best throughput of a DFG



The *loop bounds* in this example are given as $LB_{BC} = 7$ and $LB_{ABC} = 4$

The *iteration bound* is 7 -- therefore, we need at least 7 time units per iteration

Limits on Throughput



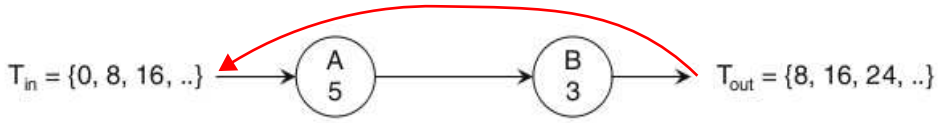
From the graph, it is clear that loop BC is the bottleneck

Note that *actors* A and C have *delay* elements on their inputs so they can operate in parallel

On the other hand, *actor* B needs to wait for the result from C before it can *fire*

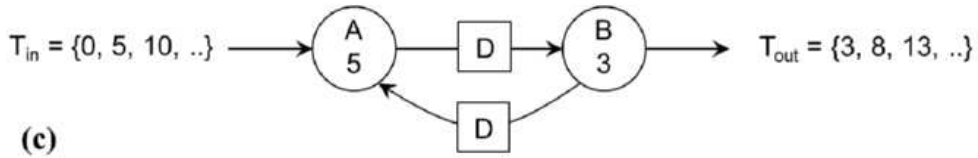
The missing *delay* element forces *actors* B and C to run sequentially

Note that *linear* graphs have implicit feedback loops that must be considered



Limits on Throughput

Also note that the iteration bound is an **upper limit** on throughput, and in reality, the DFG may **not** be able to achieve this throughput



The DFG above (from an earlier slide) has an iteration bound (5 + 3)/2 = 4 time units, but the throughput is limited to the *slowest actor* at 1 sample per 5 time units

A nice way to think about *actors* and *delays* is to consider an *actor* as a **combinational** circuit and a *delay* as a **buffer** or **pipeline** stage

Performance-Enhancing Transformations

Based on previous discussions, intuitively, it should be possible to 'tune' the DFG to enhance performance, while maintaining the same functionality

Enhancing performance either *reduces latency* or *increases throughput* or both

The following transformations will be considered:

- **Multi-rate Expansion**: A transformation which converts a multi-rate synchronous DFG to a single-rate synchronous DFG

- **Retiming**: A transformation that redistributes the *delay* elements in the DFG
  Retiming changes the throughput but does **not** change the latency or the transient behavior of the DFG

- **Pipelining**: A transformation that introduces new *delay* elements in the DFG
  Pipelining changes both the throughput and transient behavior of the DFG

- **Unfolding**: A transformation designed to increase parallelism by duplicating *actors*
  Unfolding changes the throughput but **not** the transient behavior of the DFG

Multi-rate Transformation

The following is a systematic approach to transform a *multi-rate* DFG to a *single-rate* DFG:

- Determine the PASS *firing rates* of each *actor*

- Duplicate each *actor* the number of times indicated by its *firing rate*

  For example, if *actor A* has a *firing rate* of 2, create duplicate *actors* A0 and A1

- Convert each **multi-rate** *actor* input/output to multiple **single-rate** input/outputs

  For example, an *actor* with an input consumption rate of 3 is replaced with 3 single-rate inputs

- Re-wire the *queues* in the DFG to connect all *actors*

- Re-introduce the initial *tokens* in the DFG, distributing them sequentially over the single-rate *queues*

Multi-rate Transformation

The following DFG shows *actor* A produces **three** tokens per firing, and *actor* B consumes **two** tokens per firing

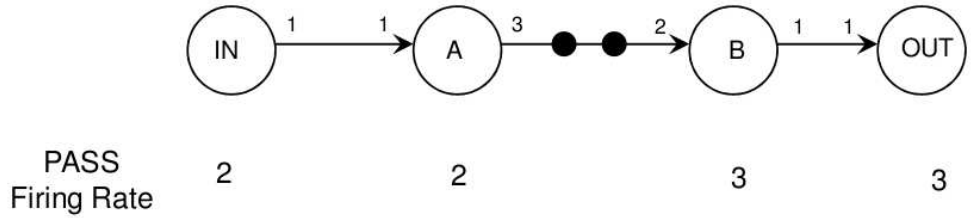
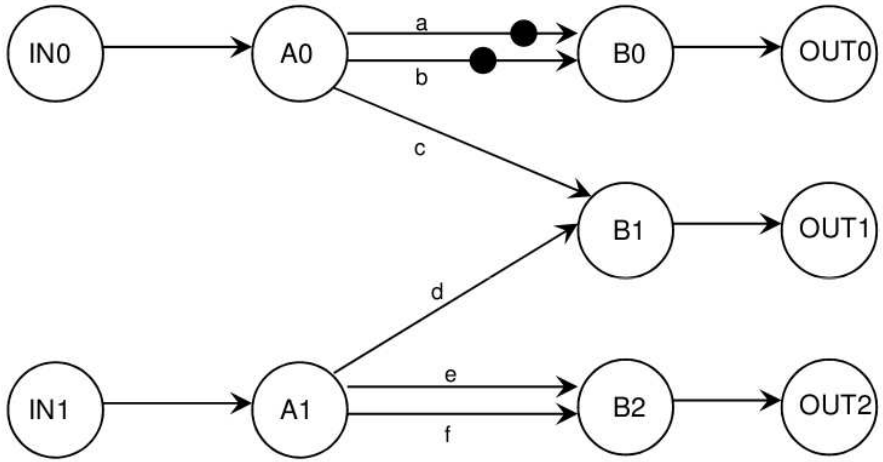


Fig. 2.28 Multi-rate data flow-graph

After completing the steps above, we obtain the following DFG



The initial tokens are redistributed in order *a*, *b*, etc

Fig. 2.29 Multi-rate SDF graph expanded to single-rate

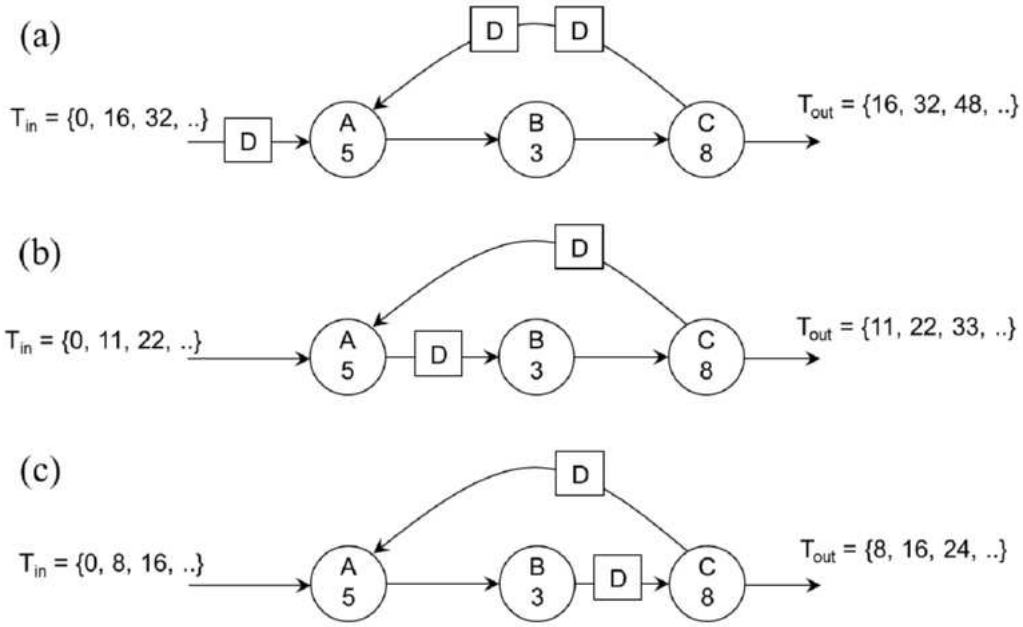
Here, the *actors* are **duplicated** according to their *firing rates*, and all *multi-rate* I/O are converted to *single-rate* I/O

Retiming Transformation

Retiming **redistributes** *delay* elements in the DFG as a mechanism to increase throughput

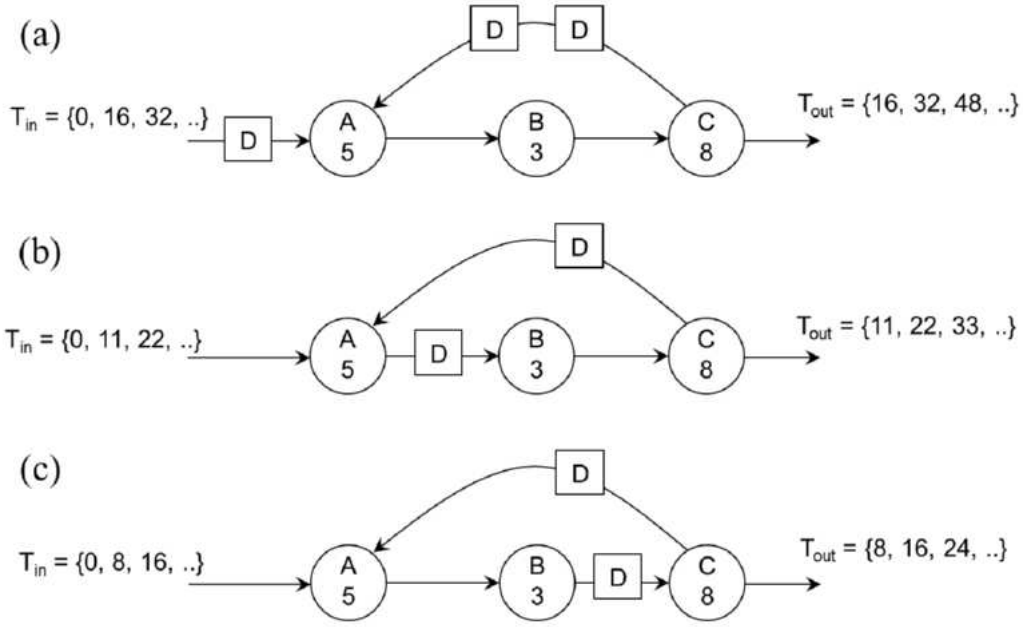
Retiming does **not** introduce new *delay* elements

Evaluation involves inspecting *successive markings* of the DFG and then selecting the one with the best performance



(a) has an iteration bound of 8 but produces data on intervals of 16 because of the sequential execution of *actors* A, B and C

Retiming Transformation



The next marking (b) is obtained by firing *actor* A, which consumes the *delay* elements on its inputs, and produces a *delay* element at its output

This functionally equivalent configuration *improves* throughput to 1 sample every 11 time units by allowing *actor* A to run in parallel with B and C

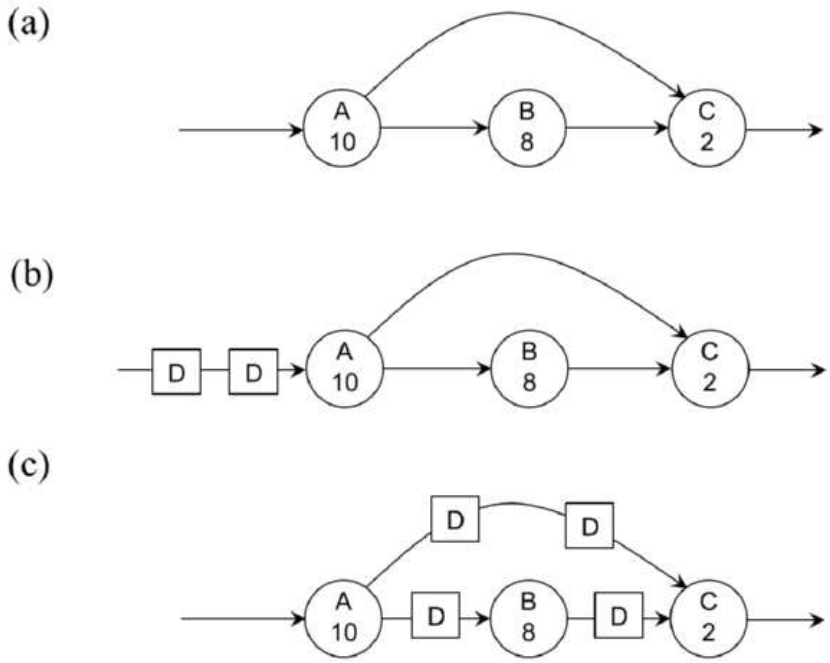
Firing B produces the next marking in (c), which achieves an *iteration bound* of 8 and represents the best that can be obtained

The last marking which fires C creates a configuration nearly equivalent to (a)

Pipelining Transformation

Pipelining increases the throughput at the cost of **increased latency**

Pipelining augments retiming with adding *delay* elements

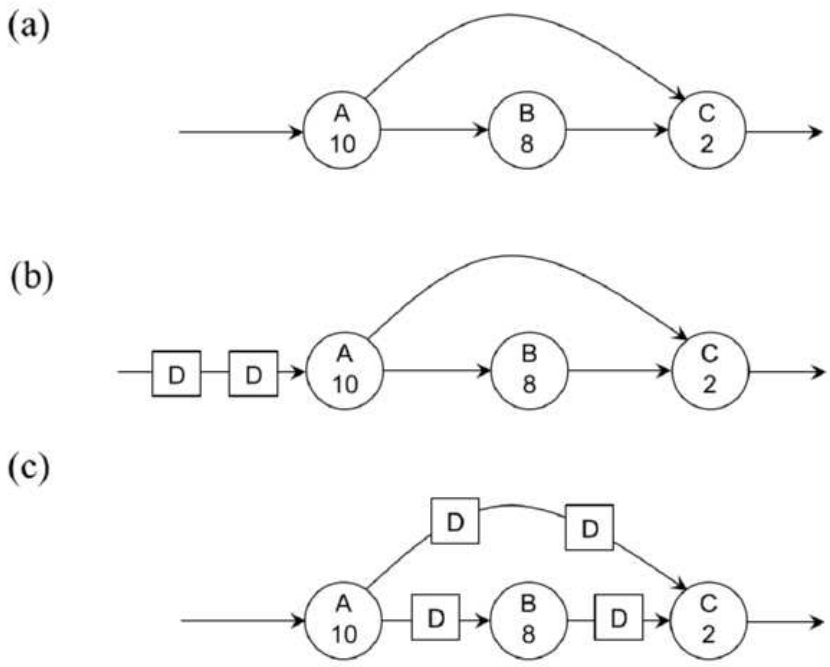


(a) is extended with two pipeline delays in (b)

Adding *delay* elements at the input increases the latency of (a) from 20 to 60

Throughput is 20, i.e., 1 sample every 20 time units

Pipelining Transformation



Retiming of the pipelined graph yields (c) after *firing* A twice and B once, which improves both throughput to 10 and latency to 20

Again we see the slowest pipeline stage determines the best achievable throughput

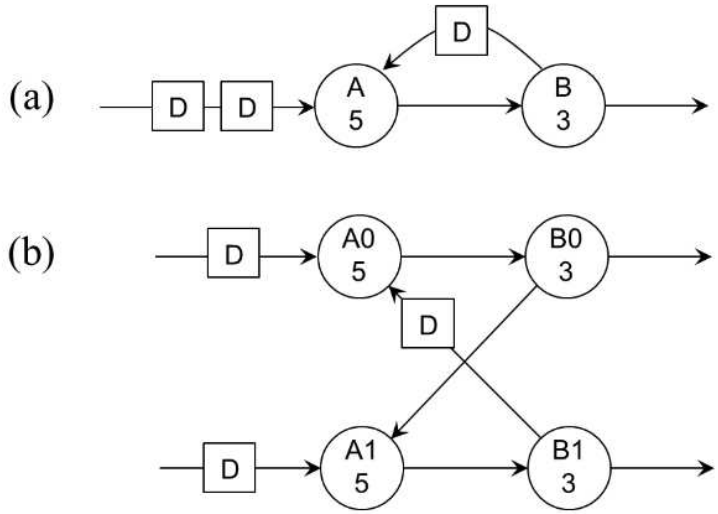
Unfolding Transformation

Unfolding is very similar to the transformation carried out for **multi-rate expansion**

Here, *actor* A in the original DFG is replicated as needed, and interconnections and *delay* elements are redistributed

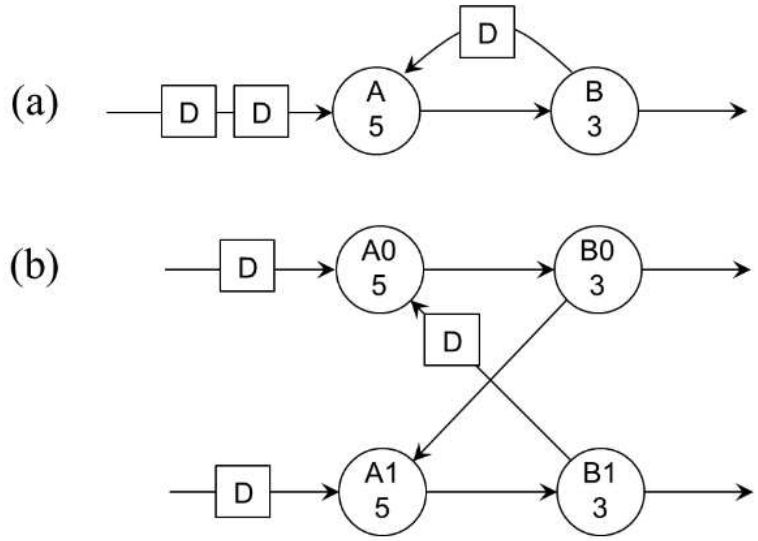
Note the original graph is *single-rate* and goal is to increase sample consumption rate

The text describes the sequence of steps that need to be applied to carry out unfolding



(a) is unfolded two times in (b), showing that the number of inputs and outputs are doubled, allowing twice as much data to be processed per iteration

Unfolding Transformation

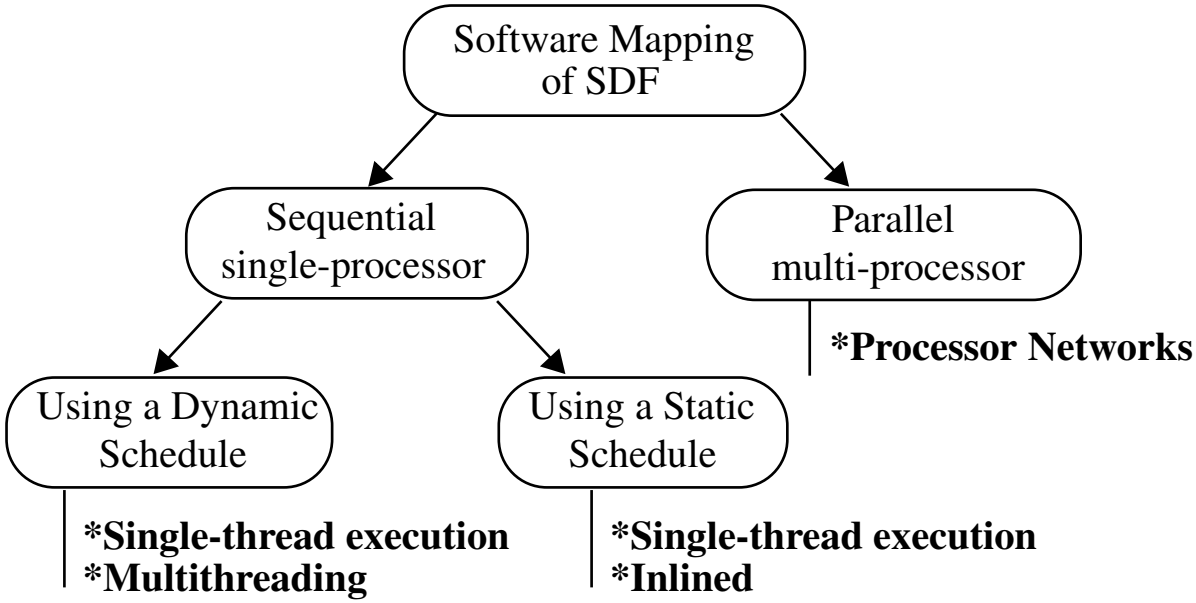


Unfolding appears to slow it down, increasing the size of the loop to include A0, B0, A1 and B1 while including only the single *delay* element

Hence, the *iteration bound* of a v-unfolded graph increases v times

Mapping DFGs to Software

There are a wide variety of approaches of mapping DFGs to software



Sequential implementations can make use of *static* or *dynamic* schedules

Parallel, multi-processor mappings require more effort due to:

- Load balancing: Mapping *actors* such that the activity on each processor is about the same
- Minimizing inter-processor communication: Mapping *actors* such that communication overhead is minimized

Mapping DFGs to Software

We focus first on *single-processor* systems, and in particular, on finding efficient versions of *sequential schedules*

As noted on the previous slide, there are two options for implementing the schedule:

- **Dynamic** schedule

  Here, software decides the order in which *actors* execute **at runtime** by testing *firing rules* to determine which *actor* can run

  Dynamic scheduling can be done in a *single-threaded* or *multi-threaded* execution environment

- **Static** schedule

  In this case, the *firing* order is determined at design time and fixed in the implementation

  The fixed order allows for a design time optimization in which the *firing* of multiple *actors* can be treated as a *single firing*

  This in turn allows for 'inlined' implementations

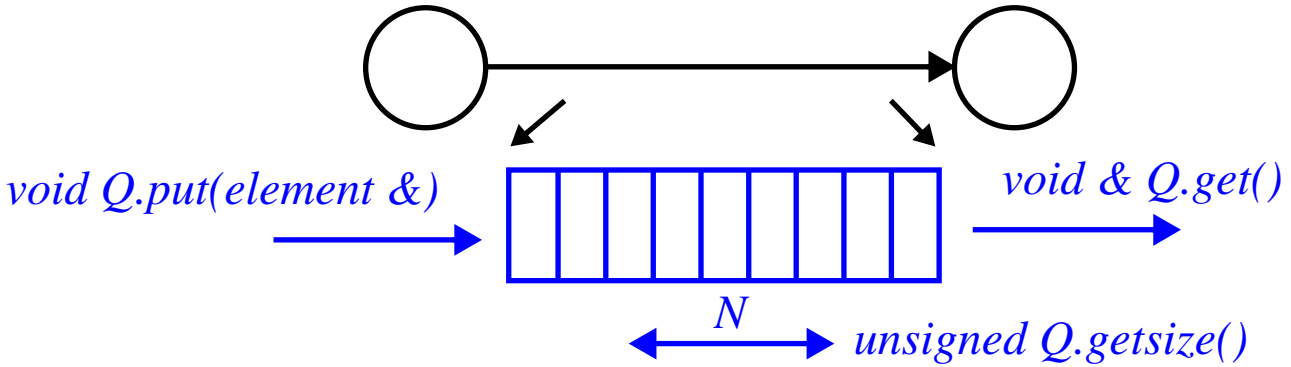
DFG Elements

Before discussing these, let's first look at C implementations of *actors* and *queues*

FIFO Queues:

Although DFGs theoretically have **infinite** length *queues*, in practice, *queues* are limited in size

We discussed earlier that constructing a PASS allows the ***maximum*** *queue* size to be determined by analyzing *actor* firing sequences



A typical software interface of a FIFO *queue* has two parameters and three methods

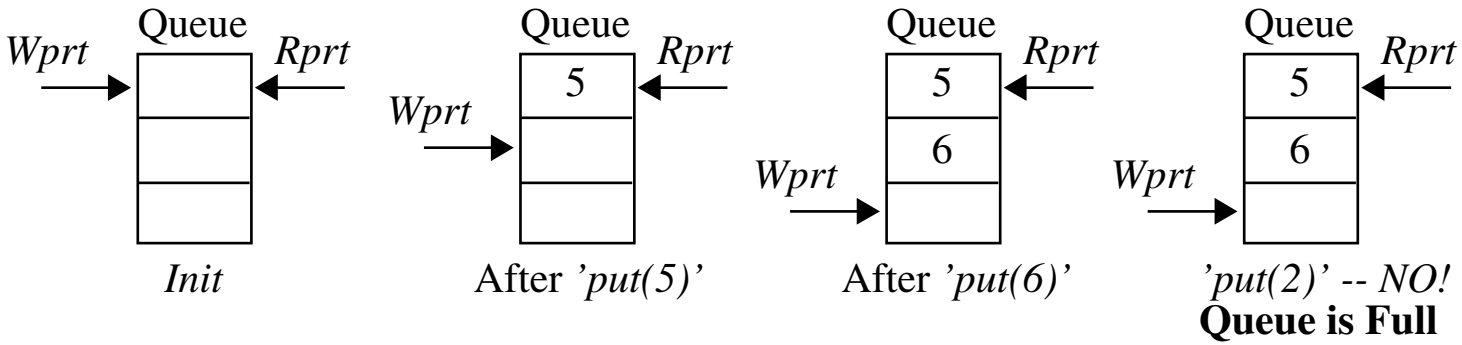
- The **# of elements** *N* that can be stored in the *queue* and the **data type** of the elements
- Methods that **put** elements into the *queue*, **get** elements from the *queue*, and **query** the current size of the *queue*

DFG Elements

Queues are well defined (standardized) data structures

A ***circular queue*** consists of an *array*, a *write-pointer* and a *read-pointer*

They use *modulo* addressing, e.g., the *Ith* element is at position (*Rptr* + *I*) *mod* *Q.getsize()*



Example *fifo* data structure definition in C:

```
#define MAXFIFO 1024

typedef struct fifo {
   int data[MAXFIFO]; // array
   unsigned wptr;     // write pointer
   unsigned rptr;     // read pointer
} fifo_t;
```

DFG Elements

```
void init_fifo(fifo_t *F); // These functions defined
void put_fifo(fifo_t *F, int d); // in text
int get_fifo(fifo_t *F);
unsigned fifo_size(fifo_t *F);

int main()
   {
   fifo_t F1;
   init_fifo(&F1);   // resets wptr, rptr
   put_fifo(&F1, 5);
   put_fifo(&F1, 6);
   printf("%d %d\n", fifo_size(&F1), get_fifo(&F1));
   printf("%d\n", fifo_size(&F1)); // prints 1
   }
```

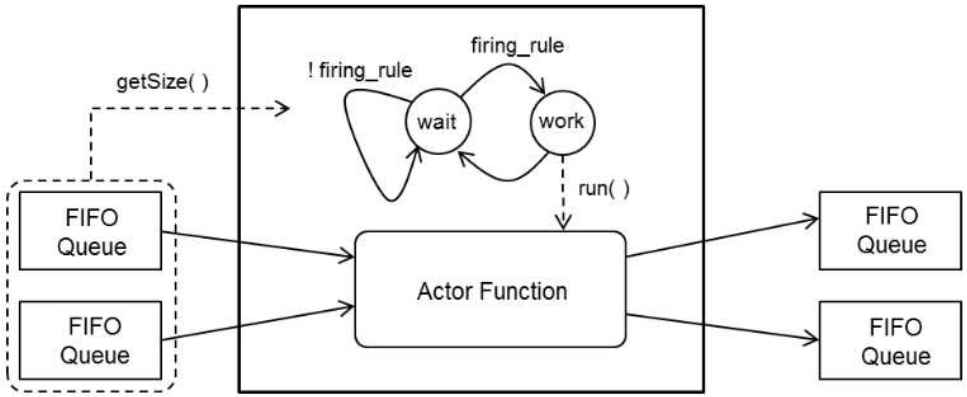
Note that the *queue* size is fixed here at compile time

Alternatively, *queue* size can be changed dynamically at runtime using *malloc()*

DFG Elements

Actors:

An *actor* can be represented as a C function, with an interface to the FIFOs



The *actor* function incorporates a finite state machine (FSM), which checks the *firing rules* to determine whether to execute the *actor* code

The *local controller* (FSM) of an *actor* has two states

wait state: start state which checks the *firing rules* immediately after being invoked by a scheduler

work state*: wait* transitions to *work* when *firing rules* are satisfied

The *actor* then reads tokens, performs calculation and writes output tokens

Example C Implementation of DFG

An example which supports up to 8 inputs and outputs per *actor*:

```
#define MAXIO 8
typedef struct actorio {
   fifo_t *in[MAXIO], *out[MAXIO];
} actorio_t;
```

An example *actor* implementation:

```
void fft2(actorio_t *g)
   {
   int a, b;
   if( fifo_size(g->in[0]) >= 2 ) // Firing rule check
      {
      a = get_fifo(g->in[0]);
      b = get_fifo(g->in[0]);
      put_fifo(g->out[0], a+b);
      put_fifo(g->out[0], a-b);
      }
   }
```

Mapping DFGs to Single Processors: Dynamic Schedule

In a dynamic system schedule, the *firing rules* of the *actors* are tested at runtime

In a single-thread dynamic schedule, we implement the **system scheduler** as a function that instantiates ALL *actors* and *queues*

The scheduler typically calls the *actors* in a *round-robin* fashion

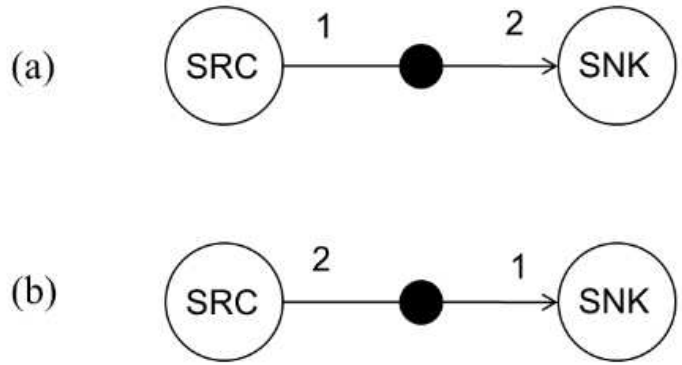
```
void main() {
   fifo_t q1, q2;
   actorio_t fft2_io = {{&q1}, {&q2}};
   ...
   init_fifo(&q1);
   init_fifo(&q2);
   while (1)
      {
      fft2_actor(&fft2_io);
      // .. call other actors
      }
   }
```

Mapping DFGs to Single Processors: Dynamic Schedule

Note that it is **impossible** to call the *actors* in the **wrong** order

This is true b/c each of them checks a *firing rule* that prevents them from running when there is no data available

An interesting question is 'is there a call order of the *actors* that is best?'



System Schedule

```
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
  }
}
```

The schedule on the right shows that ***snk*** in (a) is called as often as ***src***

However, ***snk*** will only *fire* on even numbered invocations

(b) shows a problem that is **not** handled by static schedulers

Round-robin scheduling in this case will eventually lead to *queue* overflow

Mapping DFGs to Single Processors: Dynamic Schedule

The underlying problem with (b) is that the implemented *firing rate* **differs** from the *firing rate* for a PASS, which is given as (***src***, ***snk***, ***snk***)

There are two solutions to this issue:

- Adjust the system schedule to match the PASS

```
void main()
    {
    ..
    while (1) {
        src_actor(&src_io);
        snk_actor(&snk_io);
        snk_actor(&snk_io);
        }
    }
```

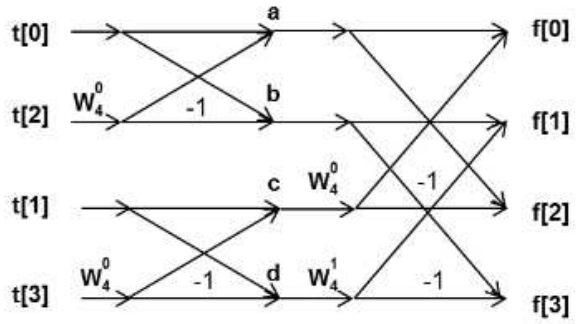
Unfortunately, this solution defeats one of the goals of a dynamic scheduler, i.e., that it automatically ***converges*** to the PASS *firing rate*

Mapping DFGs to Single Processors: Dynamic Schedule

- A better solution is to add a **while** loop to the ***snk*** *actor* code to allow it to continue execution while there are *tokens* in the *queue*

```
void snk_actor(actorio_t *g) {
  int r1, r2;
  while ((fifo_size(g->in[0]) > 0)) {
    r1 = get_fifo(g->in[0]);
    ... // do processing
  }
}
```

Mapping DFGs to Single Processors: Example Dynamic Schedule



(a)

```
a    = t[0] + W(0,4) * t[2] = t[0] + t[2]
b    = t[0] - W(0,4) * t[2] = t[0] - t[2]
c    = t[1] + W(0,4) * t[3] = t[0] + t[3]
d    = t[1] - W(0,4) * t[3] = t[1] - t[3]
f[0] = a    + W(0,4) * c    = a    + c
f[1] = b    + W(1,4) * d    = b    - j.d
f[2] = c    - W(0,4) * c    = a    - c
f[3] = b    - W(1,4) * d    = b    + j.d
```

(b)

Let's implement the *4-point Fast Fourier Transform* (FFT) shown above using a dynamic schedule

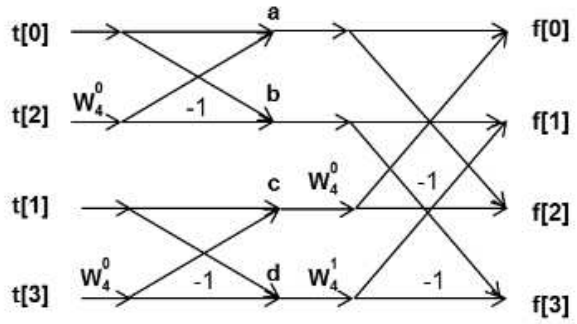
The array *t* stores 4 (time domain) samples

The array *f* will be used to store the frequency domain representation of *t*

The FFT utilizes *butterfly operations* to implement the FFT, as defined on the right side in the figure

The *twiddle* factor *W(k, N)* is a complex number defined as $e^{-j2\pi k/N}$, with *W(0, 4)* = 1 and *W(1, 4)* = *-j*

Mapping DFGs to Single Processors: Example Dynamic Schedule

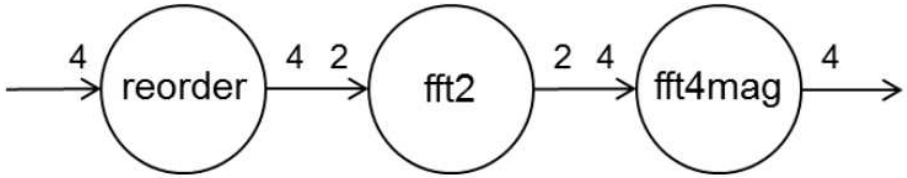


(a)

```
a    = t[0] + W(0,4) * t[2] = t[0] + t[2]
b    = t[0] - W(0,4) * t[2] = t[0] - t[2]
c    = t[1] + W(0,4) * t[3] = t[0] + t[3]
d    = t[1] - W(0,4) * t[3] = t[1] - t[3]
f[0] = a    + W(0,4) * c    = a    + c
f[1] = b    + W(1,4) * d    = b    - j.d
f[2] = c    - W(0,4) * c    = a    - c
f[3] = b    - W(1,4) * d    = b    + j.d
```

(b)

The DFG for (a) is given as follows



- *reorder*: Reads 4 tokens and shuffles them to match the flow diagram
  The *t[0]* and *t[2]* are processed by the top butterfly and *t[1]* and *t[3]* are processed by the bottom butterfly
- *fft2*: Calculates the butterflies for the left half of the flow diagram
- *fft4mag* calculates the butterflies for the right half and produces the magnitude component of the frequency domain representation

Mapping DFGs to Single Processors: Example Dynamic Schedule

The implementation first requires a valid schedule to be computed

The *firing rate* is easily determined to be $[q_{reorder}, q_{fft2}, q_{fft4mag}]$ = **[1, 2, 1]**

```
void reorder(actorio_t *g)
   {
   int v0, v1, v2, v3;
   while ( fifo_size(g->in[0]) >= 4 )
      {
      v0 = get_fifo(g->in[0]);
      v1 = get_fifo(g->in[0]);
      v2 = get_fifo(g->in[0]);
      v3 = get_fifo(g->in[0]);
      put_fifo(g->out[0], v0);
      put_fifo(g->out[0], v2);
      put_fifo(g->out[0], v1);
      put_fifo(g->out[0], v3);
      }
   }
```

Mapping DFGs to Single Processors: Example Dynamic Schedule

```
void fft2(actorio_t *g)
    {
    int a, b;
    while (fifo_size(g->in[0]) >= 2 )
        {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        put_fifo(g->out[0], a+b);
        put_fifo(g->out[0], a-b);
        }
    }
```

Mapping DFGs to Single Processors: Example Dynamic Schedule

```
void fft4mag(actorio_t *g)
    {
    int a, b, c, d;
    while ( fifo_size(g->in[0]) >= 4 )
        {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        c = get_fifo(g->in[0]);
        d = get_fifo(g->in[0]);
        put_fifo(g->out[0], (a+c)*(a+c));
        put_fifo(g->out[0], b*b - d*d);
        put_fifo(g->out[0], (a-c)*(a-c));
        put_fifo(g->out[0], b*b - d*d);
        }
    }
```

while loops are used in all *actors* as a mechanism to deal with *mismatches* between the scheduler's calls to *actors* and their actual firing rates (as noted earlier)

Mapping DFGs to Single Processors: Example Dynamic Schedule

```
int main()
   {
   fifo_t q1, q2, q3, q4;
   actorio_t reorder_io = {{&q1}, {&q2}};
   actorio_t fft2_io = {{&q2}, {&q3}};
   actorio_t fft4_io = {{&q3}, {&q4}};

   init_fifo(&q1);
   init_fifo(&q2);
   init_fifo(&q3);
   init_fifo(&q4);

// Test vector fft([1 1 1 1])
   put_fifo(&q1, 1);
   put_fifo(&q1, 1);
   put_fifo(&q1, 1);
   put_fifo(&q1, 1);
```

Mapping DFGs to Single Processors: Example Dynamic Schedule

```
    // Test vector fft([1 1 1 0])
       put_fifo(&q1, 1);
       put_fifo(&q1, 1);
       put_fifo(&q1, 1);
       put_fifo(&q1, 0);

       while (1)
          {
          reorder(&reorder_io);
          fft2(&fft2_io);
          fft4mag(&fft4_io);
          }
       return 0;
       }
```

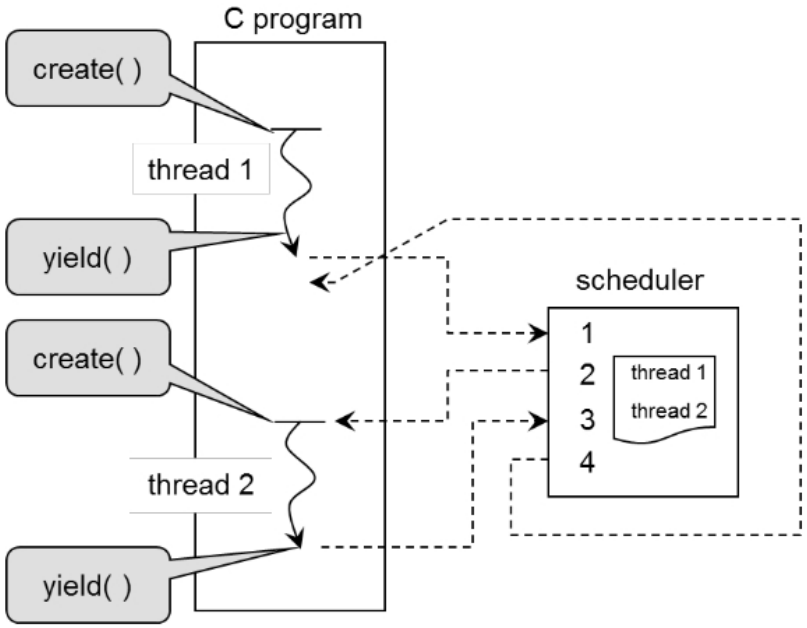
The deterministic property of SDFs and the **while** loops inside the *actors* allow the call order shown above to be re-arranged while preserving the functional behavior

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

In multi-threaded programming, each *actor* (implemented as a function) lives in a separate thread

The threads are time-interleaved by a scheduler in single processor environments

Systems in which threads **voluntarily** relinquish control back to the scheduler is referred to as ***cooperative*** multithreading



Such a system can be implemented using two functions *create()* and *yield()* as shown

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

The scheduler can apply different strategies to schedule threads, with the simplest one shown above as a *round-robin* schedule

Quickthreads is a **cooperative multithreading** library

The quickthreads API (Application Programmers Interface) consists of 4 functions

- *spt_init()*: initializes the threading system
- *spt_create(stp_userf_t *F, void *G)* creates a thread that will start execution with user function *F*, and will be passed a single argument *G*
- *stp_yield()* releases control over the thread to the scheduler
- *stp_abort()* terminates a thread (prevents it from being scheduled)

Here's an example

```
#include "../qt/stp.h"
#include <stdio.h>
```

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```
void hello(void *null)
    {
    int n = 3;
    while (n-- > 0)
        {
        printf("hello\n");
        stp_yield();
        }
    }

void world(void *null)
    {
    int n = 5;
    while (n-- > 0)
        {
        printf("world\n");
        stp_yield();
        } }
```

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```
int main(int argc, char **argv)
   {
   stp_init();
   stp_create(hello, 0);
   stp_create(world, 0);
   stp_start();
   return 0;
   }
```

To compile and execute:

```
gcc -c ex1.c -o ex1 ../qt/libstp.a ../qt/libqt.a
./ex1
hello
world
hello
world
hello
world\nworld\nworld
```

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

A multi-threaded version of the SDF scheduler, using the *fft2 actor*

```
void fft2(actorio_t *g) {
   int a, b;
   while (1)
      {
      while (fifo_size(g->in[0]) >= 2)
         {
         a = get_fifo(g->in[0]);
         b = get_fifo(g->in[0]);
         put_fifo(g->out[0], a+b);
         put_fifo(g->out[0], a-b);
         }
      stp_yield();
      }
```

Mapping DFGs to Single Processors: Multi-Thread Dynamic Schedule

```
void main()
    {
    fifo_t q1, q2, q3, q4;
    actorio_t fft2_io = {{&q2}, {&q3}};
    ...
    stp_create(fft2, &fft2_io); // create thread
    ...
    stp_start();          // start system scheduler
    }
```

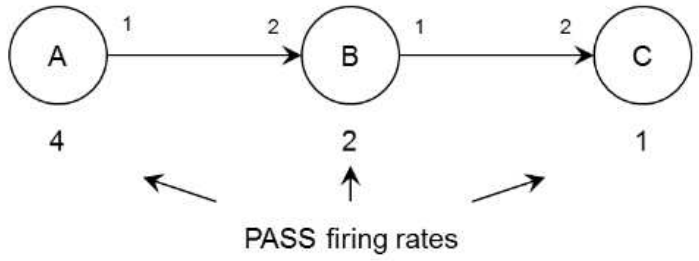
Note, as before, the *actor* code must enable convergence to the PASS *firing rate* (through while loops) in order to avoid *queue* overflow

Mapping DFGs to Single Processors: Static Schedule

From the PASS analysis of an SDF graph, we know at least one solution for a feasible sequential schedule

This solution can be used to optimize the implementation in several ways

- We can remove the *firing rules* since we know the exact sequential schedule
  This yields only a small performance benefit
- We can also determine an optimal interleaving of the *actors* to minimize the storage requirements for the *queues*
- Finally, we can create a fully **inlined** version of the SDF graph which eliminates the *queues* altogether




```
while(1) {
  // call A four times
  A(); A(); A(); A();

  // call B two times
  B(); B();

  // call C one time
  C();
}
```

Here, the relative *firing rates* of A, B, and C must be 4, 2, and 1 to yield a PASS

Mapping DFGs to Single Processors: Static Schedule

Given the interleaving schedule on the right, *queue* AB will need to store at most 4 *tokens* and *queue* BC at most 2 *tokens* in steady-state

However, the interleaving schedule (A,A,B,A,A,B,C) is better because the maximum # of tokens on any *queue* is now 2

Therefore, the schedule determined using PASS is **not** necessarily the optimal (in fact, finding the best schedule is an optimization problem)

As noted, implementing a truly static schedule means we do NOT need to check *firing rules* since the required tokens are guaranteed to be present

Consider optimizing the four-point FFT with a **single-thread** SDF system and a **static schedule**

The 3 *actors*, *reorder*, *fft2* and *fft4mag*, have firing rates 1, 2 and 1, which yields a static, cyclic schedule [*reorder*, *fft2*, *fft2*, *fft4mag*]

Software Implementation: Sequential Targets with Static Schedule

There are two simple **optimizations** that can be applied here

- The *firing schedule* is **static** and **fixed**, and therefore the access order of *queues* is also fixed

  This allows the queues to be *optimized out* and replaced with **fixed variables**

The *queue* access can be replaced as shown in the comments

```
loop {
   ...
   q1.put(value1); // replace with r1 = value1;
   q1.put(value2); // replace with r2 = value2;
   ...
   .. = q1.get(); // replace with .. = r1;
   .. = q1.get(); // replace with .. = r2;
   }
```

Software Implementation: Sequential Targets with Static Schedule

- A second optimization involves **inline**'ing the *actor* code in the main program

  In combination with the above optimization, this *eliminates* the *firing rules* and reduces the entire dataflow graph to a **single** function

```
void dftsystem(int in0, in1, in2, in3,
   *out0, *out1, *out2, *out3) {
   int reorder_out0, reorder_out1;
   int reorder_out2, reorder_out3;
   int fft2_0_out0, fft2_0_out1;
   int fft2_0_out2, fft2_0_out3;
   int fft2_1_out0, fft2_1_out1;
   int fft2_1_out2, fft2_1_out3;
   int fft4mag_0_out0, fft4mag_0_out1;
   int fft4mag_0_out2, fft4mag_0_out3;

// Reorder operation
   reorder_out0 = in0; reorder_out1 = in2;
   reorder_out2 = in1; reorder_out3 = in3;
```

Software Implementation: Sequential Targets with Static Schedule

```
// Two fft2 implementations
fft2_0_out0 = reorder_out0 + reorder_out1;
fft2_0_out1 = reorder_out0 - reorder_out1;
fft2_1_out0 = reorder_out2 + reorder_out3;
fft2_1_out1 = reorder_out2 - reorder_out3;

// fft4 implementation
fft4mag_out0 = (fft2_0_out0 + fft2_1_out0) *
               (fft2_0_out0 + fft2_1_out0);
fft4mag_out1 = (fft2_0_out1 * fft2_0_out1) -
               (fft2_1_out1 * fft2_1_out1);
fft4mag_out2 = (fft2_0_out0 - fft2_1_out0) *
               (fft2_0_out0 - fft2_1_out0);
fft4mag_out3 = (fft2_0_out1 * fft2_0_out1) -
               (fft2_1_out1 * fft2_1_out1);
```

Software Implementation: Sequential Targets with Static Schedule

These optimizations reduce the runtime of the program significantly

For example, we have eliminated testing of the *firing rules* and calls to the *queue* and *actor* functions

This is possible here because a *valid PASS* could be determined from the DFG, as well as *fixed schedule* to implement the PASS

Note that we have traded some of the **runtime flexibility** for **improved efficiency**

Mapping DFGs to Hardware

As indicated in previous lectures, SDF graphs are particularly well-suited for mapping directly into hardware

For SDFs with *single-rate* schedules, all *actors* can execute in parallel using the same clock

The rules for mapping *single-rate* SDFs to hardware are straightforward

- All *actors* map to a single *combinational* hardware circuit
- All FIFO *queues* map to wires (without storage)
- Each initial token in a *queue* is replaced with a register

This means that two *actors* with no token on the *queue* between them will be effectively represented as two back-to-back combinational circuits

Timing requirements regarding the delay of the combinational logic must be met, i.e., all computation within the combinational logic **must complete in 1 clock cycle**

Therefore, in practice, the length of the *actor* sequence will be limited

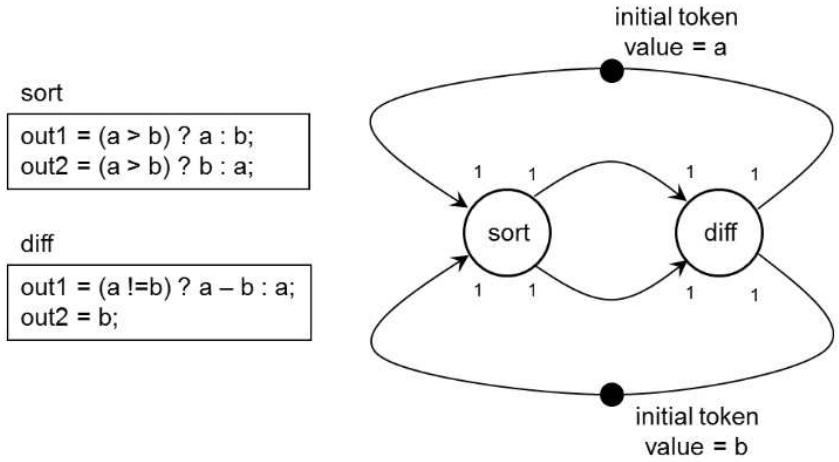
Mapping DFGs to Hardware

The performance annotations we've used in DFGs are only estimates of delay

Obtaining precise combinational circuit delays falls in the realm of CAD tools

We will treat the delay annotations for an *actor* as the **critical path delay**, i.e., the delay of the worst-case path in the *actor's* combinational circuit

Let's use Euclid's Greatest Common Divisor algorithm to illustrate the process



Euclid's GCD Algorithm

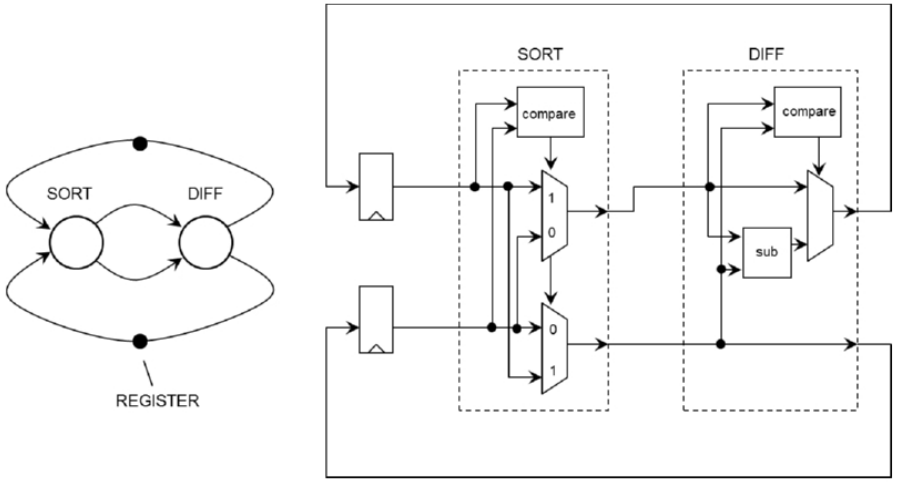
The *sort actor* reads the two initial token values *a* and *b*, sorts them and reproduces them on the output

The *diff actor* subtracts the smallest number from the largest one as long as they are not equal

Mapping DFGs to Hardware

You should convince yourself that a PASS exists using the techniques described earlier for SDFs

Following the mapping rules given above, we obtain the following circuit



The *sort* and *diff* actors are implemented using a comparator, a subtractor plus a few multiplexers

Note that the delay through the circuits of both *actors* define the maximum achievable clock frequency

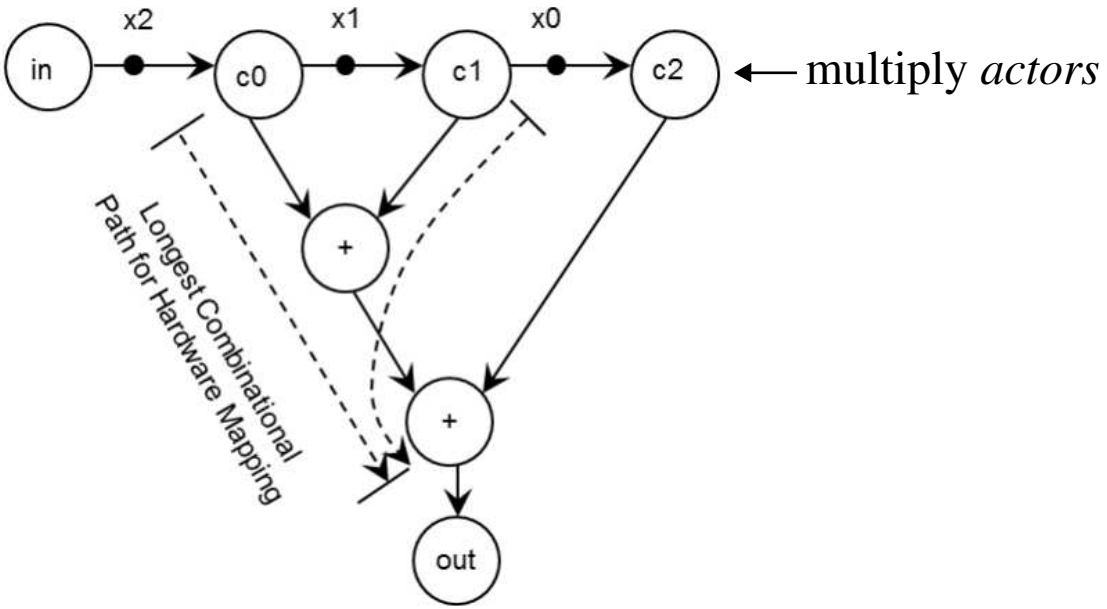
If the critical path delays are 5 and 15 ns, then max. operating freq. is 50 MHz

Mapping DFGs to Hardware: Pipelining

The **pipelining** transformation discussed earlier is a very popular technique to deal with path delays that limit clock frequency

Consider a dataflow specification of a **digital filter**

The filter computes a *weighted sum* of the samples arriving from the input stream as $out = x0.c2 + x1.c1 + x2.c0$

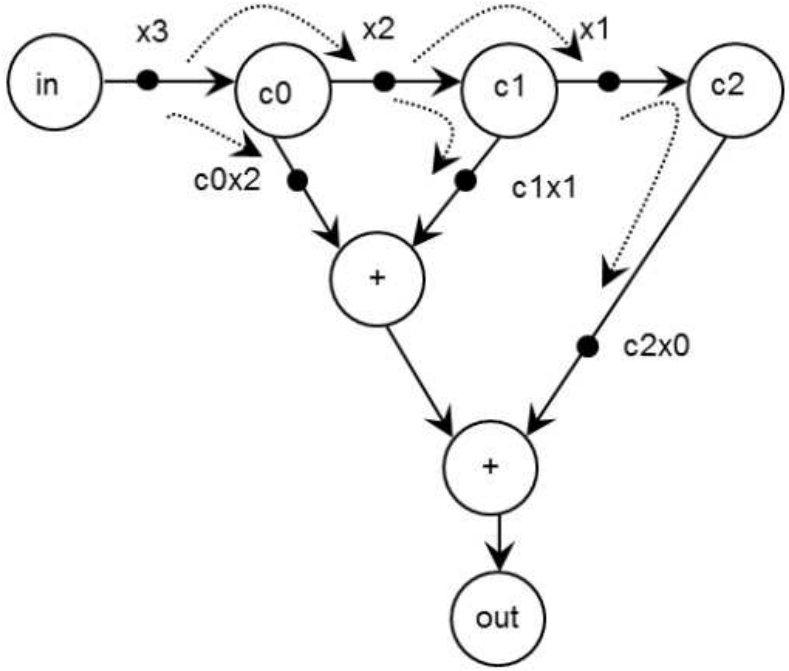


The critical path of this graph is associated with the one of the multiplier *actors*, *c0* or *c1*, in series with two addition *actors*

Mapping DFGs to Hardware: Pipelining

Pipeling allows additional tokens to be added (at the expense of increase latency)

Here, one is introduced by the *in* actor



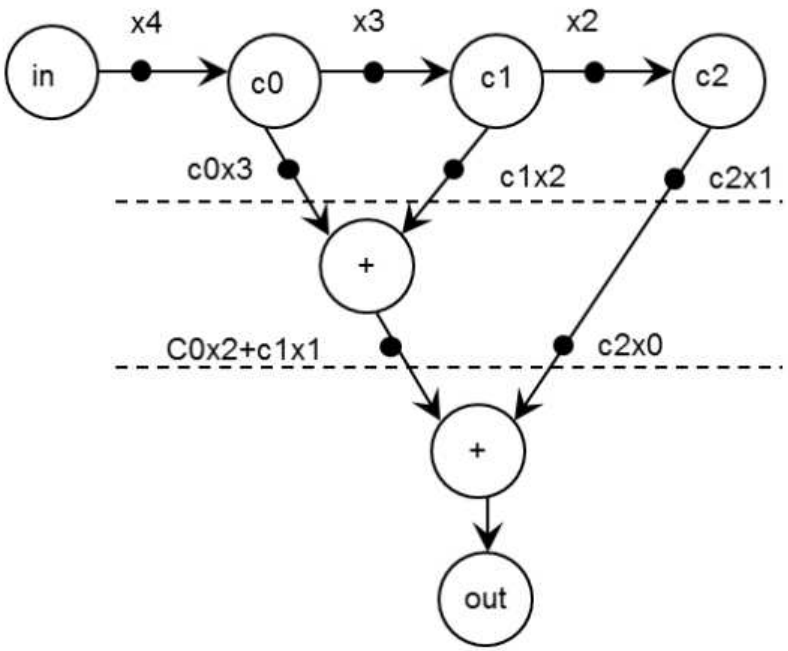
And then retiming is used to redistribute the tokens, as shown by the above marking, which is produced after *firing c0*, *c1* and *c2*

Retiming creates additional registers and an additional pipeline stage

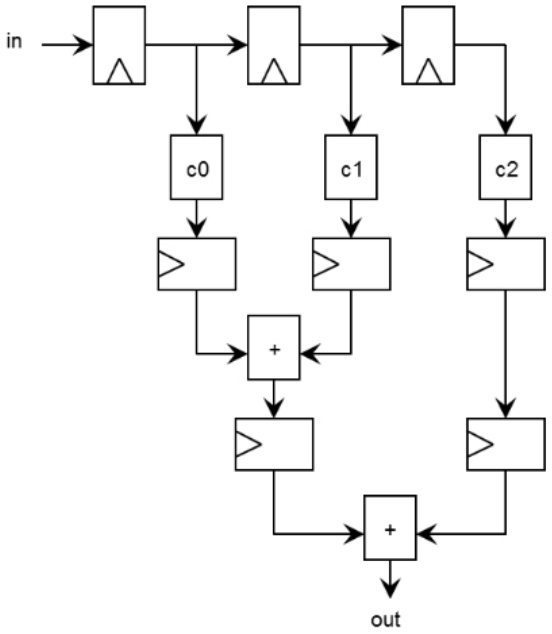
The critical path is now reduced to two back-to-back adder *actors*

Mapping DFGs to Hardware: Pipelining

This final marking is obtained by allowing the *in actor* to add one more token and then using retiming to *fire c0*, *c1*, *c2* and the top add *actor*



The resulting **pipelined** DFG implementation

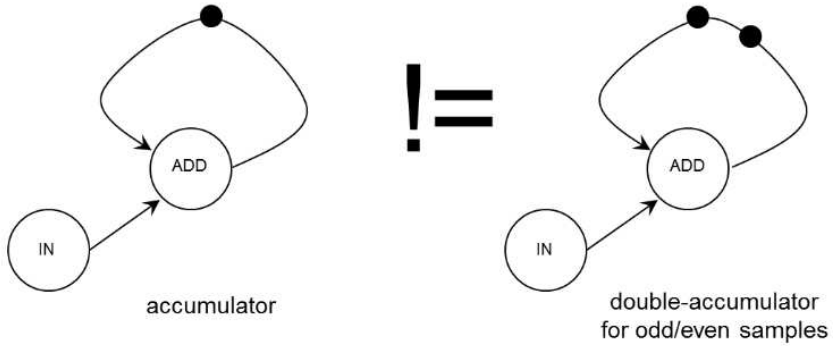


The schematic on the right shows a fully pipelined implementation

Note that it is **not** possible to introduce *arbitrary* initial tokens in a graph *without* following the actor's *firing* rules

Doing so would likely change the behavior of the system

Mapping DFGs to Hardware: Pipelining



This change in behavior is obvious in the case of feedback loops, as shown here for an *accumulator* circuit

- Using a **single** *token* in the feedback loop of an add *actor* will accumulate all input samples, as shown on the left
- Using **two** *tokens* in the feedback loop will accumulate the odd samples and even samples separately

When pipelining a DFG, be sure to follow the rules for subsequent markings

When adding new *tokens*, add them only at the input or output of the DFG, outside of any loops

And then use retiming to redistribute the *tokens* to reduce critical path delay

Control and Data Edges

C programs (and pseudo-code) are often used as prototypes because they represent **high-level descriptions** of system behavior

However, C is sequential and cannot be directly mapped into parallel hardware

Nonetheless, codesigners must develop skills to carry out this task (it is listed as one of our course objectives)

A solid understanding of C program structure and the relationships that exist between C operations is foundational to this process

In this lecture, we consider two fundamental relationships between C operations

- **Data edge**: is a relationship between operations where data produced by one operation is consumed by another
- **Control edge**: is a relationship between operations that relates to the order in which the operations are performed

Control and Data Edges

Consider the following example that returns the max of *a* or *b*

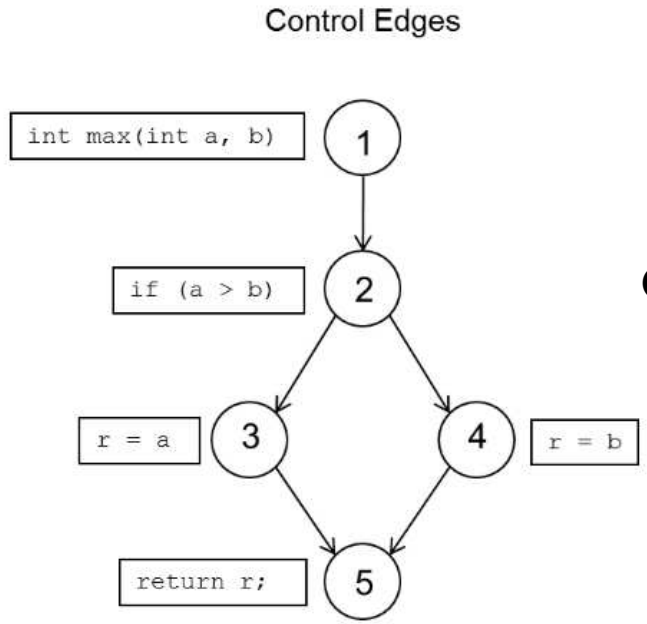
```
int max(int a, b) // operation 1 - enter the function
   {
   int r;
   if (a > b )     // operation 2 - if-then-else
      r = a;       // operation 3
   else
      r = b;       // operation 4
   return r;       // operation 5 - return max
   }
```

As you can see, our analysis treats each of the C statements as individual operations

To find the control edges in this program, we need to identify all possible paths through this program

Control and Data Edges

For example, operation 2 will always execute *after* operation 1



Control Flow Graph (CFG)

We can use a **control flow graph** (CFG) to capture this relationship, by adding a directed edge between these operations (which are represented as bubbles)

The *if-then-else* operation includes two out-going edges to represent each of the two execution paths

Control and Data Edges

The **data flow graph** (DFG) is constructed by analyzing the data production (writes) and consumption (reads) patterns for each of the variables

```
int max(int a, b) { // operation 1 - produce a, b
  int r;
  if (a > b )    // operation 2 - consume a, b
    r = a;       // operation 3 - consume a and (a>b),
                 //               produce r
  else
    r = b;       // operation 4 - consume b and (a>b),
                 //               produce r
  return r;      // operation 5 - consume r
}
```

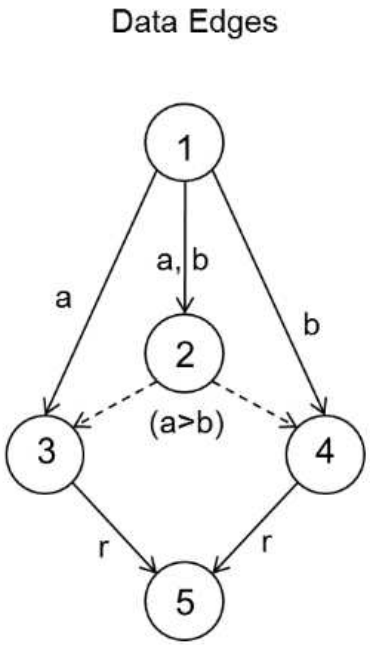
Data edges are added between operations which write and then read a variable

For example, operation 1 defines (writes) the values of *a* and *b*

Variable *a* is read by operation 2 and 3 while *b* is read by operation 2 and 4

This produces data edges from 1 to 2 for *a* and *b*, and to 3 for *a* and 4 for *b*

Control and Data Edges



Data Flow Graph (DFG)

Control statements in C also generate data edges

For example, the *if-then-else* statement evaluates a *flag* ($a > b$), which reads a and b

The boolean *flag carries* the value of ($a > b$) from operation 2 to operations 3 and 4

Note that unlike CFGs, edges in DFGs are labeled with a specific variable

Implementation Issues

CFGs and DFGs capture the behavior of the C program graphically

This leads naturally to the following question:

*What are the important parts of a C program that MUST be preserved in **any** implementation of that program?*

• Data edges reflect requirements on the flow of information

Important note: If you change the flow of data, **you change the meaning of the algorithm**

• Control edges, on the other hand, provide a nice mechanism to break down the algorithm into a sequence of operations (a recipe)

They are **not** fundamental to preserving correct functional behavior in an implementation

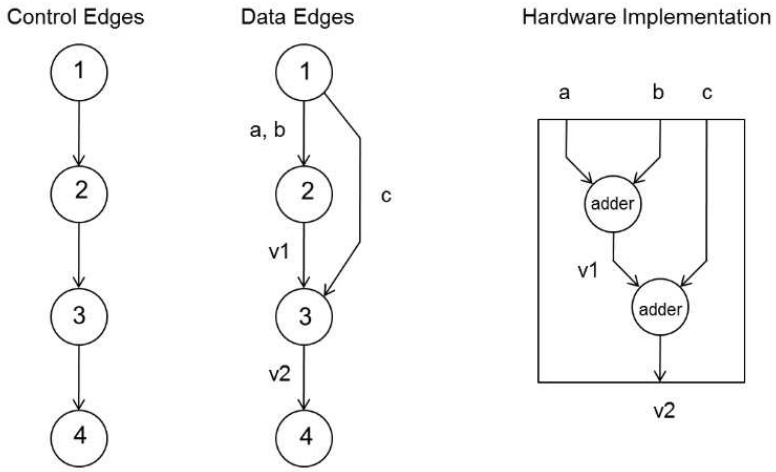
It follows then that **data edges** MUST be preserved while **control edges** can be removed and/or manipulated

Implementation Issues

Parallelism in the underlying architecture can be leveraged to remove control edges, e.g., superscalar processors can execute instructions *out-of-order*

On the other hand, parallel architectures MUST always preserve data dependencies otherwise, the results will be erroneous

```
int sum(int a, b, c) {          // operation 1
   int v1;
   v1 = a + b;                  // operation 2
   v2 = v1 + c;                 // operation 3
   return v2; }                 // operation 4
```



A fully parallel hardware implementation of this program can in fact carry out both additions in a *single clock cycle*

The sequential order specified by the CFG is eliminated in the hardware implementation

Construction of the Control Flow Graph

Let's define a systematic method to convert a C program to a CFG assuming:

- Each **node** in the graph represents a single operation (or C statement)
- Each **edge** of the graph represents an execution order for the two operations connected by that edge

Since C executes sequentially, this conversion is straightforward in most cases

The only exception occurs when multiple control edges originate from a single operation

Consider the *for loop* in C

```
for (i = 0; i < 20; i++) {
    // body of the loop
}
```

This statement includes four distinct parts:

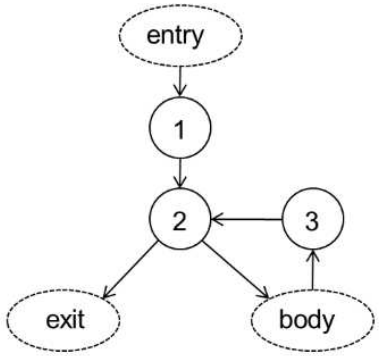
- loop initialization
- loop condition
- loop-counter increment operation
- body of the loop

Construction of the Control Flow Graph

The *for loop* introduces three nodes to the CFG

for loop contributes multiple operations →

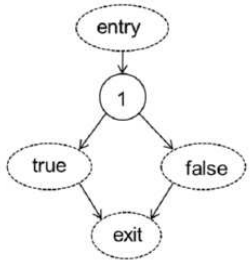
```
      (1)    (2)    (3)
for (i=0; i < 20; i++) {
   // body of the loop
}
```



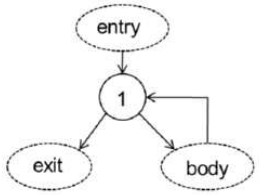
Dashed components, *entry*, *exit* and *body*, are other CFGs of the C program which have ***single-entry*** and ***single-exit*** points

The *do-while* loop and the *while-do* loop are similar iterative structures

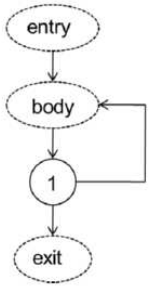
```
(1)
if(a < b) {
   // true branch
} else {
   // false branch
}
```




```
(1)
while (a < b) {
   // loop body
}
```



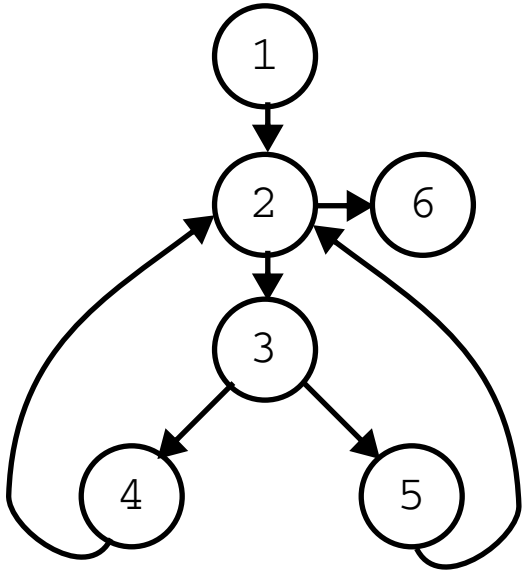

```
do {
  // loop body
} while (a<b)
        (1)
```



Construction of the Control Flow Graph

Consider the CFG for the GCD algorithm.

```
1: int gcd (int a, int b) {
2:     while (a != b) {
3:         if (a > b)
4:             a = a - b;
           else
5:             b = b - a;
           }
6:     return a;
       }
```



A **control path** is defined as a sequence of control edges that traverse the CFG

For example, each *non-terminating* iteration of the while loop will follow the path 2->3->4->2 or else 2->3->5->2

Control paths are useful in constructing the DFG

Construction of the Data Flow Graph

Let's also define a systematic method to convert a C program to a DFG assuming

- Each **node** in the graph represents a single operation (or C statement)
- Each **edge** of the graph represents a data dependency

Note that the CFG and the DFG will contain the **same set of nodes** -- only the edges will be different

While it is possible to derive the DFG directly from a C program, it is easier to create the CFG **first** and use it to derive the DFG

The method involves tracing control paths in the CFG while simultaneously identifing corresponding read and write operations of the variables

Our analysis focuses on C programs that do NOT have *arrays* or *pointers*

Text includes discussion and examples on how to handle these more complex data structures

Construction of the Data Flow Graph

Ad-hoc method:

- Start at the node where a variable is read (which is referred to as a *read-node*)
- Identify the CFG nodes that assign to that variable (referred to as *write-nodes*)
- Introduce a data edge between a read and write node under the condition that the *control path* does **NOT** pass through another *write-node* for that variable
- Repeat for all read nodes

This procedure identifies all data edges related to *assignment statements*, but **not** those originating from *conditional expressions* in control flow statements

However, these data edges are easy to find

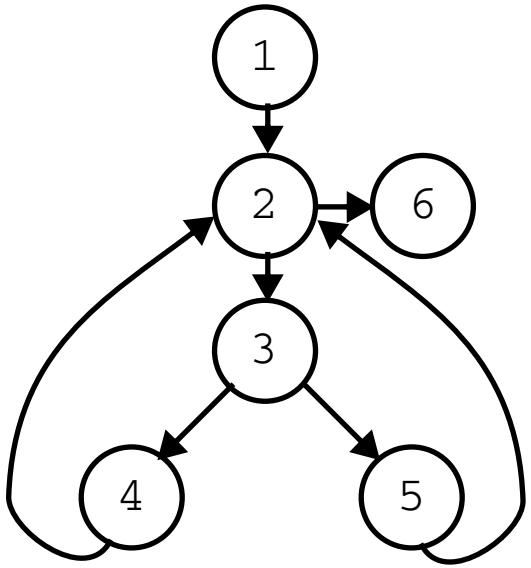
They originate from the condition evaluation and **affect all** the operations whose execution depends on that condition

Let's derive the DFG of the GCD program

We first pick a node where a variable is read

Construction of the Data Flow Graph

```
1: int gcd (int a, int b) {
2:     while (a != b) {
3:         if (a > b)
4:             a = a - b;
           else
5:             b = b - a;
           }
6:     return a; }
```



Consider stmt 5:

There are two variable-reads in this statement, one for *a* and one for *b*

Consider *b* first

Find all nodes that reference *b* by **tracing backwards** through predecessors of node 5 in the CFG -- this produces the ordered sequence 3, 2, 1, 4, and 5

Both nodes 1 and 5 write *b* and there is a *direct path* from 1 to 5 (e.g. 1, 2, 3, 5), and from 5 to 5 (e.g. 5, 2, 3, 5)

Therefore, we need to add data edges for *b* from 1 to 5 and from 5 to 5

Construction of the Data Flow Graph

A similar process can be carried out for variable-read of *a* in node 5

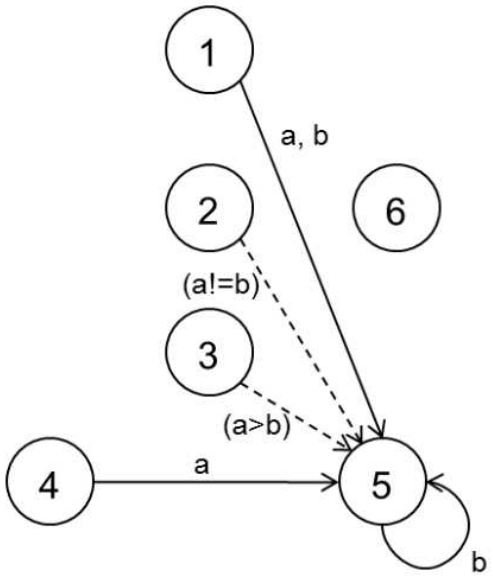
Nodes 1 and 4 write into *a* and there is a direct control path from 1 to 5 and from 4 to 5

Hence, data edges are added for *a* from 1 to 5 and from 4 to 5

To complete the set of data edges into node 5, we also need to identify all **conditional expressions** that affect the outcome of node 5

From the CFG, node 5 depends on the condition evaluated in node 3 ($a > b$) AND the condition evaluated in node 2 (a != b)

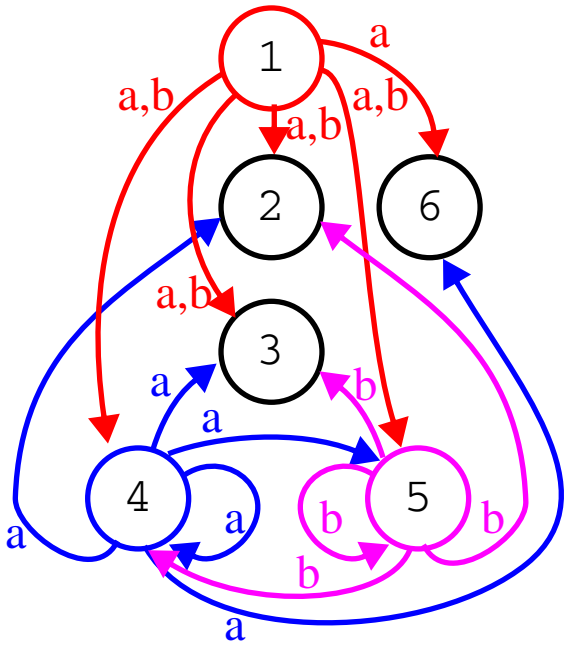
Partial DFG for node 5



Construction of the Data Flow Graph

The final DFG for all nodes and all *variable-reads* for GCD is shown below.

```
1: int gcd (int a, int b) {
2:     while (a != b) {
3:         if (a > b)
4:             a = a - b;
           else
5:             b = b - a;
           }
6:     return a; }
```



Note: this DFG leaves out the data edges originating from conditional expressions

Being able to abstract a complex C program to a DFG is essential for codesign

Translating C to Hardware

As mentioned, control and data flow analysis can be helpful in translating C into hardware

Translating data structures and pointers from C to hardware can get tricky

For the purpose of this course, we restrict our analysis as follows

- Only **scalar** C code is used (no pointers, arrays or other data structures)
- We assume each C statement executes in a single clock cycle

We first create the CFG and DFG for the C program

The control edges translate to signals that control *datapath operations*

The data edges define the *interconnection* of the datapath components

Translating C to Hardware: Data Path Design

Data Path Design: The following process can be used to create the datapath

• Each variable in the C program is translated into a register and a multiplexer

The multiplexer is used to allow the register to be written to from any one of *multiple sources*

The *select inputs* of the multiplexor are connected to the **controller**

The default multiplexer setting is to *preserve the register contents*, which means the output of the register is fed back to the input

• For each node (operation) in the DFG, create the corresponding combinational circuit from the C expression

For example, the expression *b* - *a* is used for the operation `a = b - a;` which is implemented using a subtractor

Note that **conditional expressions** also generate datapath elements whose outputs define *flags* used by the hardware controller

Translating C to Hardware: Data Path Design

- The datapath and the registers are connected consistent with the DFG

  Assignments connect combinational circuit outputs to register inputs, while the **data edges** connect register outputs to combinational circuit inputs

  System I/O is connected to datapath inputs and register outputs resp.

Let's convert the GCD program to a hardware implementation

- The variables *a* and *b* are assigned to registers
- The conditional expressions for the *if* and *while stmts* require an equality- and greater-than comparator circuit
- The subtractions *b* - *a* and *a* - *b* are implemented using subtractors

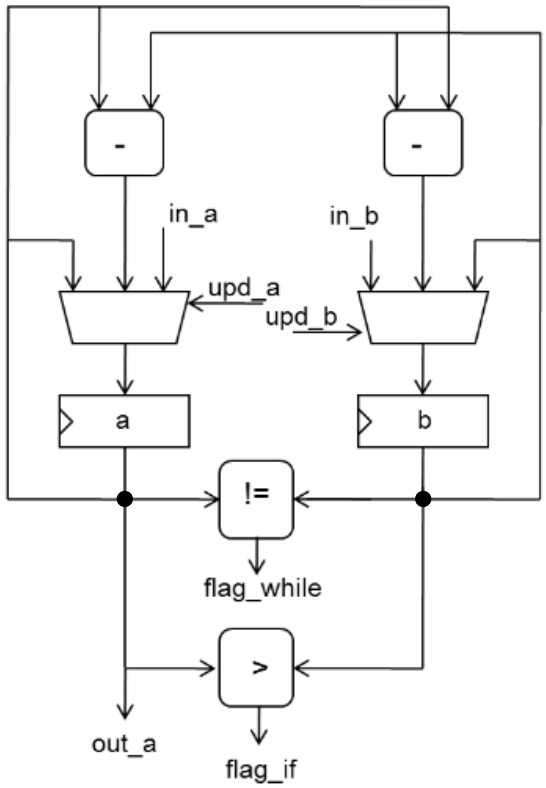
The connectivity of the components is defined by the data edges of the DFG

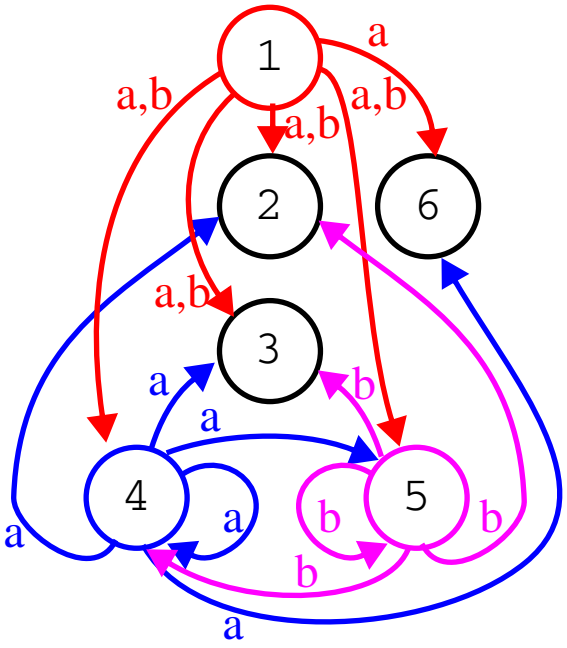
The resulting datapath has **two data inputs** (*in_a* and *in_b*), and one data output (*out_a*)

The circuit needs **two control variables**, *upd_a* and *upd_b* (outputs of controller) and it produces two **flags**, *flag_while* and *flag_if* (inputs to controller)

Translating C to Hardware: Data Path Design

```
1: int gcd(int a, int b) {
2:   while (a != b) {
3:     if (a > b)
4:       a = a - b;
       else
5:       b = b - a;
     }
6:   return a;
   }
```





The directed edges in the DFG correspond to the connections in the schematic

Schematic representations of a circuit are **low-level representations** with lots of detail (similar to assembly code in programming languages)

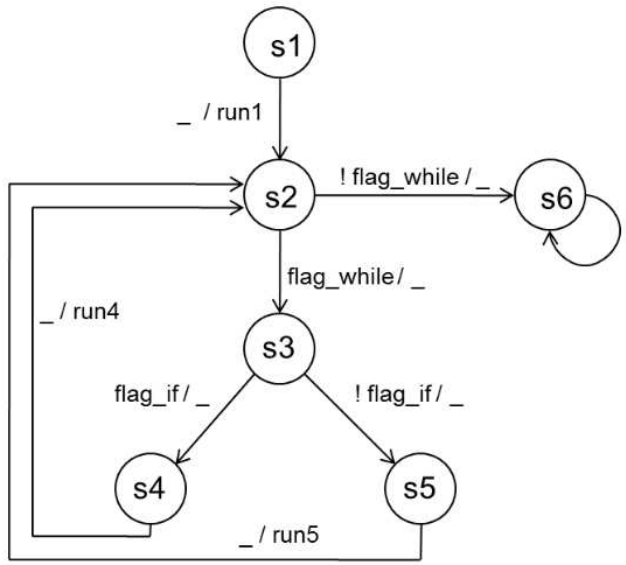
We will learn how to create HLS and behavioral VHDL descriptions that synthesize to schematics similar to the one shown here

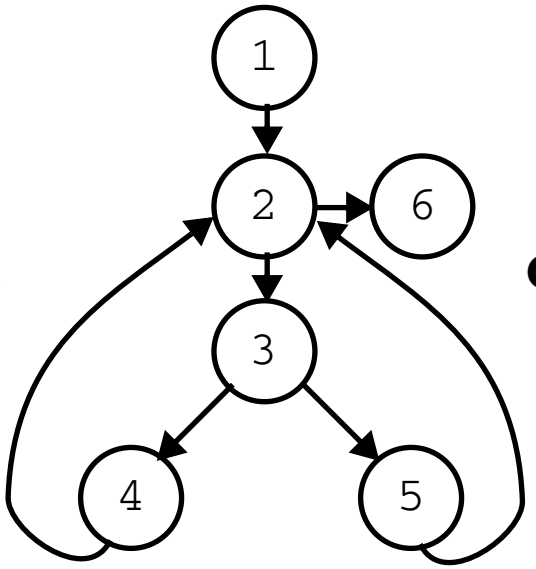
Translating C to Hardware: Control Path Design

Controller Design: The design of the controller can be derived directly from the CFG and translated into a *finite state machines* (**FSM**)

A FSM is typically depicted using bubbles and directed edges, similar to CFGs

FSM





CFG

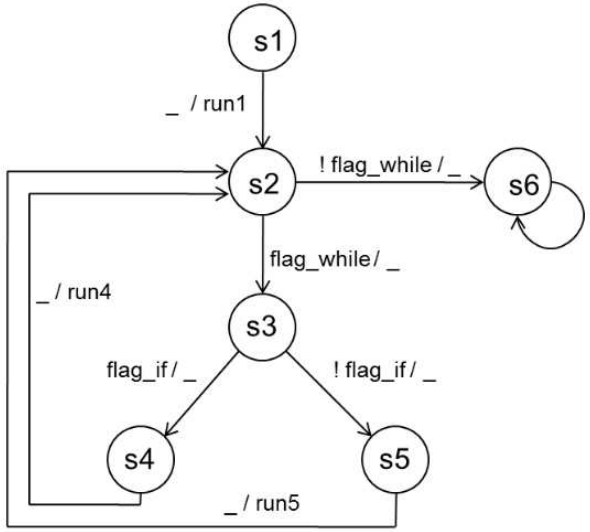
Unlike CFGs, the edges in FSMs are labeled with ***condition/command*** tuples

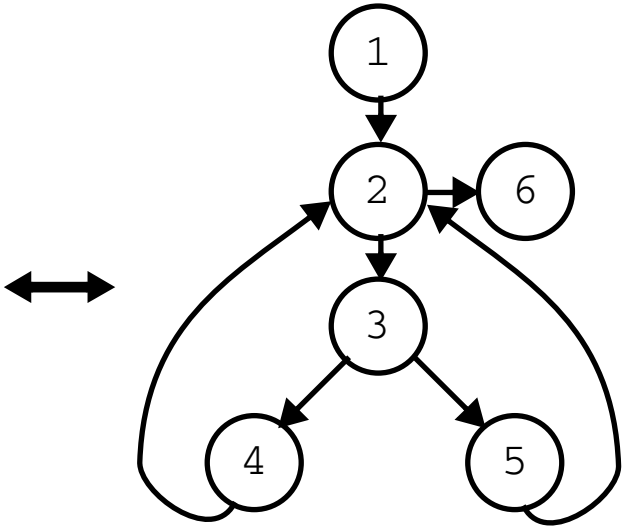
The '_' in *_/run1* means don't care, i.e., the transition is unconditional

The command component is given by the symbol *run1*, which is used to control the data path and therefore represents an **output** of the FSM

Translating C to Hardware: Control Path Design

FSM





CFG

Similarly, *flag_while/_* means the transition out of the current state is ***conditional*** on *flag_while*

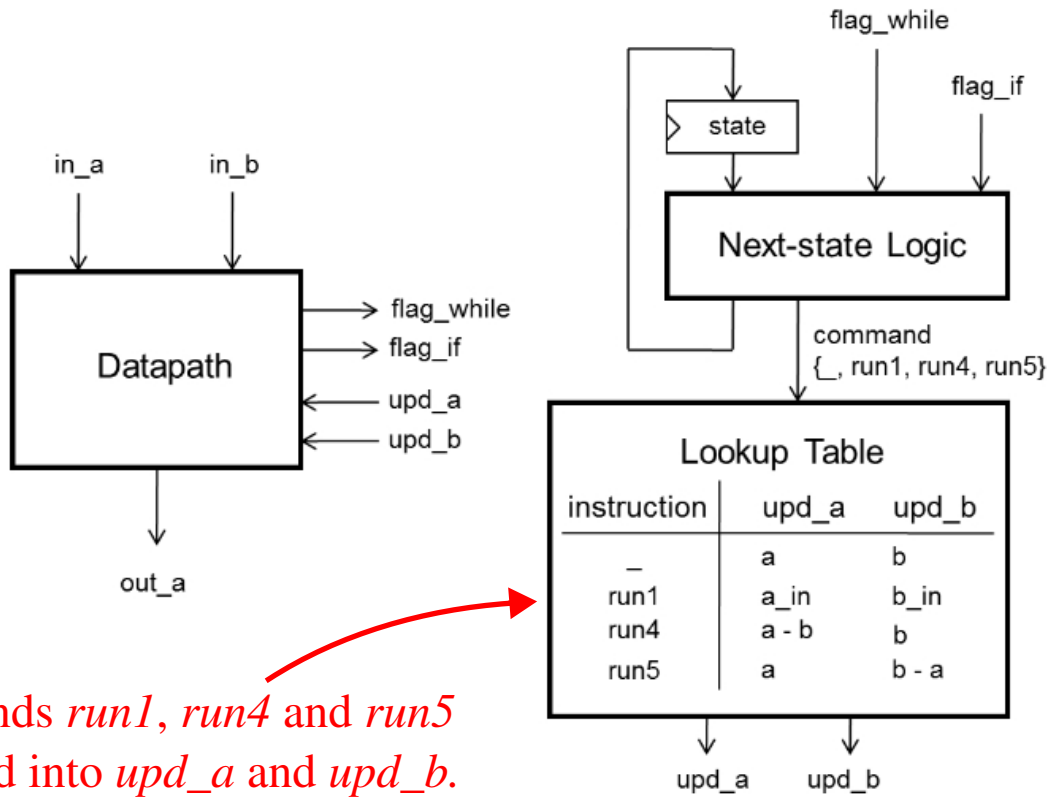
And the command '_' is a **hold** operation, which means maintain the current state of the datapath and registers

The command set for this FSM includes _, *run1*, *run4*, *run5*

These symbols will be used to create the *upd_a* and *upd_b* data path control signals

Translating C to Hardware: Control Path Design

Hardware implementation of the GCD controller with data path



The commands *run1*, *run4* and *run5* are decoded into *upd_a* and *upd_b*.

One each clock cycle, the controller generates a new command based on the current state and the current values of *flag_while* and *flag_if*

The combination of the data path and controller is referred to as a ***finite state machine with datapath*** (**FSMD**)

Translating C to Hardware: Example

FSMDs are central to custom hardware design, so we discuss them further in the next chapter (and throughout this course)

The table shows an example execution, where each row of the table corresponds to **one clock cycle**:

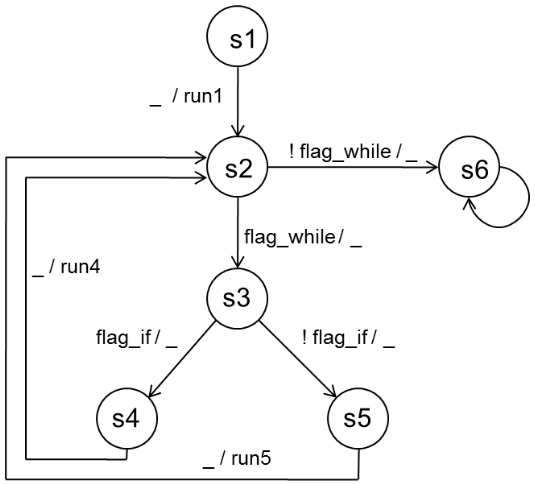


Table 3.1 Operation of the hardware to evaluate GCD(4,6)

Cycle	a	b	State	flag_if	flag_while	Next State	upd_a	upd_b
1	_	_	s1	_	_	s2	in_a	in_b
2	6	4	s2	1	1	s3	a	b
3	6	4	s3	1	1	s4	a	b
4	6	4	s4	1	1	s2	a-b	b
5	2	4	s2	0	1	s3	a	b
6	2	4	s3	0	1	s5	a	b
7	2	4	s5	0	1	s2	a	b-a
8	2	2	s2	0	0	s6	a	b
9	2	2	s6	_	_	s6	a	b

Note that this solution is sub-optimal, in paricular:

- The resulting implementation **limits parallelism** -- it executes a single C statement per clock cycle and does **not share** datapath operators

  For example, only one subtractor is needed in the implementation because only one is ever used in any given clock cycle

Single-Assignment Programs

Converting into hardware with **one** C-stmt/clock is not very efficient

This *one cycle-per-statement* is similar to what microprocessors do when they execute a program

A more lofty goal is to devise a **translation strategy** that allows the execution of multiple C stmts/clock

But our original **variable-to-register** mapping strategy creates a performance bottleneck

This is true because only **one storage location** exists for each variable and therefore, sequential updates to it will each require one clock cycle

We fix this problem by converting the C code to a **single-assignment program**

This is done by creating new variables for each sequential assignment stmt

Single-Assignment Programs

Consider a simple example:

```
a = a + 1;
a = a * 3;
a = a - 2;
```

Our previous strategy requires 3 clock cycles to execute these statements

Let's re-write this as:

```
a2 = a1 + 1;
a3 = a2 * 3;
a4 = a3 - 2;
```

This code allows *a2* and *a3* to be mapped to wires and *a4* to a register, reducing the clock cycle count to 1

Single-Assignment Programs

Note: care must be taken that all assignments are taken into account, which might be difficult to determine

```
a = 0;
for (i = 1; i < 6; i++)
  a = a + i;
```

After conversion to single-assignment, it remains unclear what version of *a* should be read inside of the loop

```
a1 = 0;
for (i = 1; i < 6; i++)
  a2 = a + i; // which version of a to read
```

The answer is that both *a1* and *a2* are needed and it depends on the iteration, i.e., *a1* is needed on the first iteration and *a2* on subsequent iterations

Single-Assignment Programs

The solution is to introduce a new **merge** variable that *selects* from the two versions that are available

```
a1 = 0;
for (i=0; i<5; i++) {
    a3 = merge(a1, a2);  // merge two instances.
    a2 = a3 + 1;
}
```

In a hardware implementation, the *merge* operation is mapped into a **multiplexer**, with the *selection signal* derived from the test of (i == 0)

Using these transformations, we can reformulate any program into single assignment form

Single-Assignment Programs

Consider the GCD program

```
int gcd (int a, int b) {
   while (a != b)
      {
      if (a > b)
         a = a - b;
      else
         b = b - a;
      }
   return a;
   }
```

Single-Assignment Programs

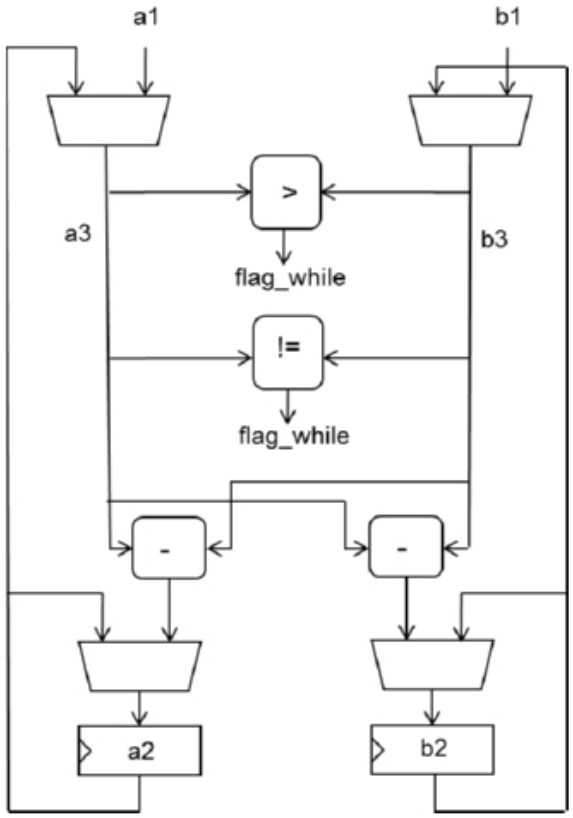
The equivalent *single-assignment form*:

```
int gcd (int a1, int b1)
    {
    while ((a3 = merge(a1, a2)) != (b3 = merge(b1, b2)))
        {
        if (a3 > b3)
          a2 = a3 - b3;
        else
          b2 = b3 - a3;
        }
    return a2;
    }
```

Single-Assignment Programs

The implementation of this single-assignment version might look like:

merge operations implemented as multiplexers →



Here, *a2* and *b2* are mapped into registers while the other variables are replaced with wires

This type of manipulation allows multiple C statements to be executed per clock

FSM

In this lecture, we begin looking at the options for mapping algorithms into hardware

We begin at the behavioral level of abstraction with a **register-transfer-level** (RTL) description of hardware using VHDL

At the heart of RTL descriptions is the *finite state machine with datapath* (**FSMD**)

The *finite state machine* component is constructed (usually by the synthesis tools) to realize the *control flow* components of the algorithm

The *datapath* is synthesized into a set of mathematical hardware operations, e.g., add and multiply, that process the data

We begin with algorithms that are largely characterized as control-only algorithms

We consider full-blown FSMD in future lectures

Our goal is to become proficient at translating software algorithms (and other high-level specifications) into hardware implementations

FSM

An FSM is designed to control the execution of a sequence of operations over multiple clock cycles, which makes it ideal for emulating software execution

It usually incorporates decision constructs, including *if-else* and *case* statements, which are used extensively in software execution for implementing control flow

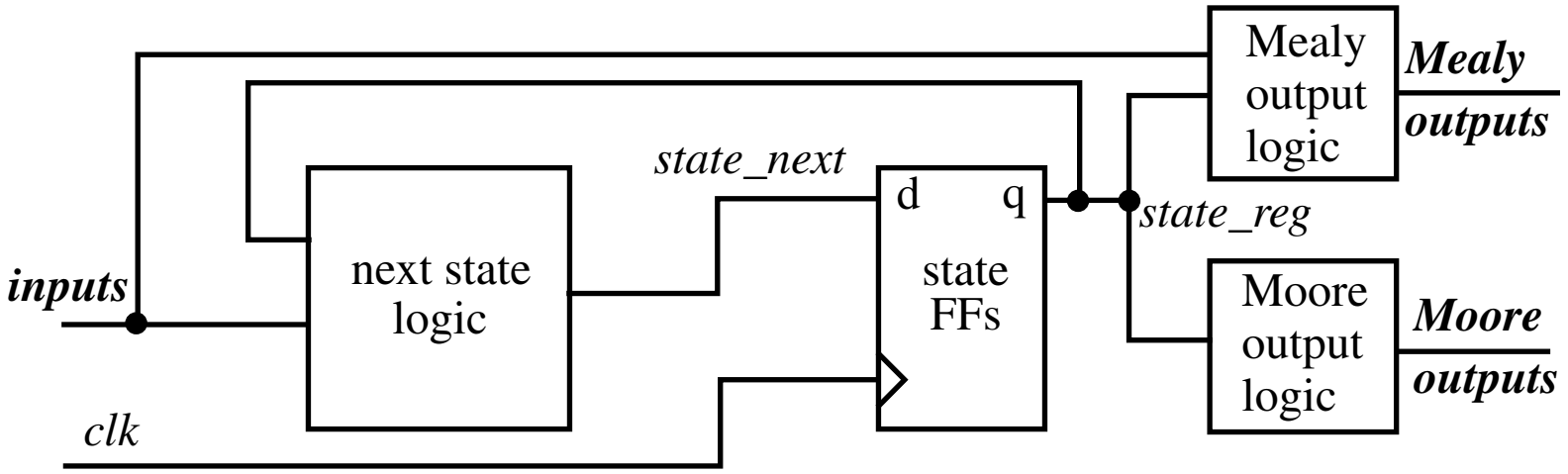
Advanced FSMs can be designed to implement pipelines, recursion, out-of-order execution and exception handling

An FSM is a sequential digital machine, which is defined using:

- A set of states
- A set of inputs and outputs
- A state transition function
- An output function

FSM

The Mealy form of an FSM (as you've seen in previous classes) allows the *inputs* to effect both the *next state logic* and *outputs*



Both Mealy and Moore outputs can be present in an FSM, and usually are

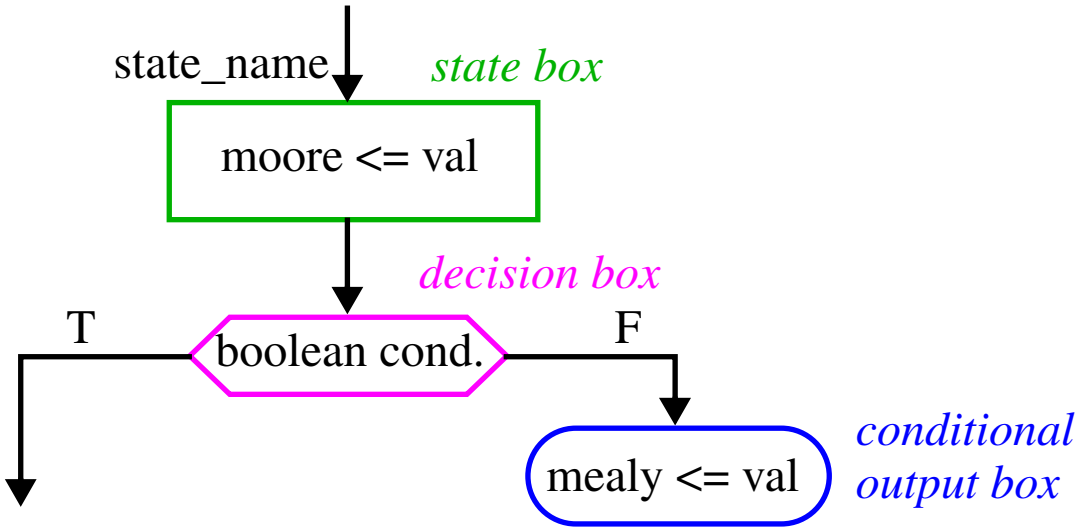
Moore outputs depend only on the current state

VHDL descriptions of the FFs and combinational logic (*next state logic* and *output logic*) are separated into two **process** blocks using two segment style

ASM

State diagrams are the most common graphical representation of FSMs, but algorithmic state machine (ASMs) diagrams can also be used

ASMs provide a more explicit representation of control and data path elements



Each *state box* has only one exit and is usually followed by a *decision box*

Conditional output boxes can only follow *decision* boxes and are used to define the values of Mealy outputs as a function of the Boolean conditions

EVERYTHING that follows a state box (to the next state) is **combinational logic** that is *active* in the current clock cycle

Secure Memory Access Controller

Let's consider a control algorithm that is designed to provide a secure access control mechanism to an on-chip memory

We discussed the architecture and provided instruction on how to create a project with **GPIO** and **BRAM** IP blocks in the laboratory screencasts

The general purpose I/O is configured to implement two memory-mapped 32-bit registers located between the PS and PL sides of the Zynq SoC

The BRAM is configured as a stand-alone memory and is embedded within the PL side

Our memory controller's primary function is to allow C programs running on the ARM microprocessor to load and unload the BRAM in a restricted manner

High Level Algorithm for a Secure Memory Access Controller

Let's start with at level of abstraction of the FSM running in the hardware, e.g.,

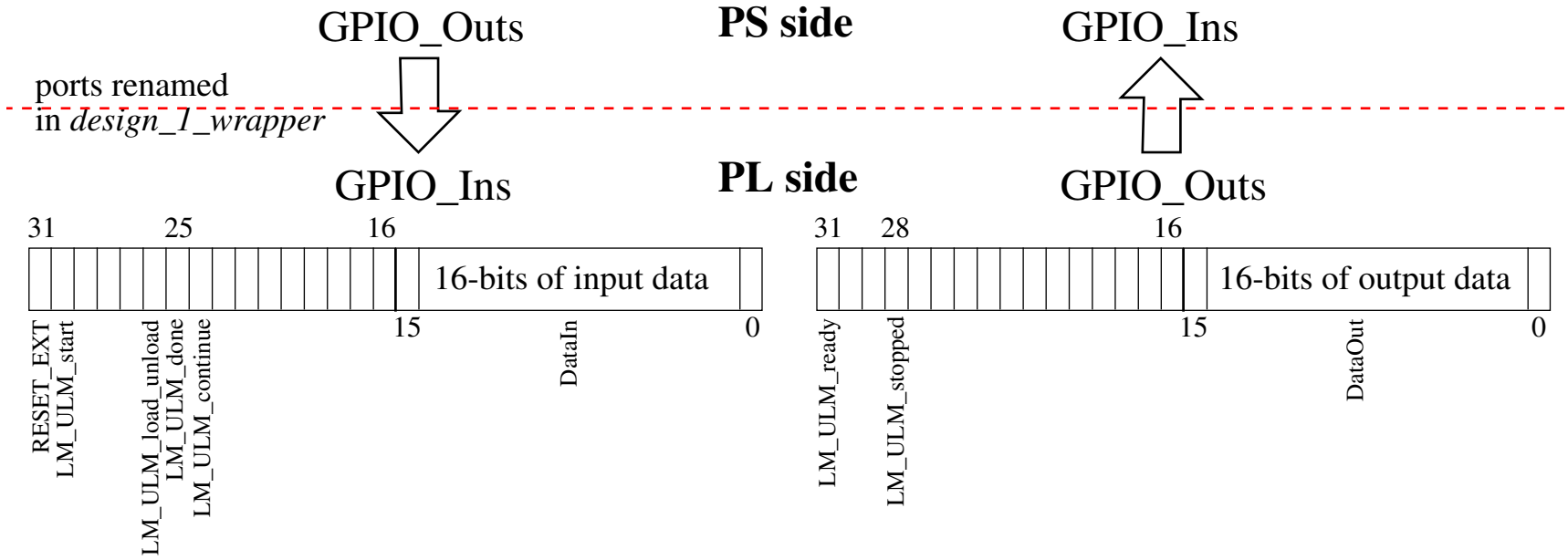
```
if ( 'start' == 1 )
   store 'base addr' and 'upper limit'
   while ('base addr' != 'upper limit' and 'done' == 0 )
      if ( 'load_unload' == 0 )
         PNL_BRAM['base addr'] = GPIO
      else
         GPIO = BRAM_PNL['base addr']
      'base addr' = 'base addr' + 1
```

One of the first issues we'll need to deal with is setting up a communication mechanism between the C program and the FSM

The GPIOs are visible to both the C program and the VHDL, and will be used for this purpose

GPIO Register Definitions for Secure Memory Access Controller

We will define the bit-fields within the two 32-bit GPIO registers as follows:



Some of the high order 16-bits are designated for **control** while all of the lower order 16-bits are designated for **data** transfers

Note that the GPIOs cross clock domains, with the PL side running at 50 or 100 MHz and the PS side running at more than 600 MHz

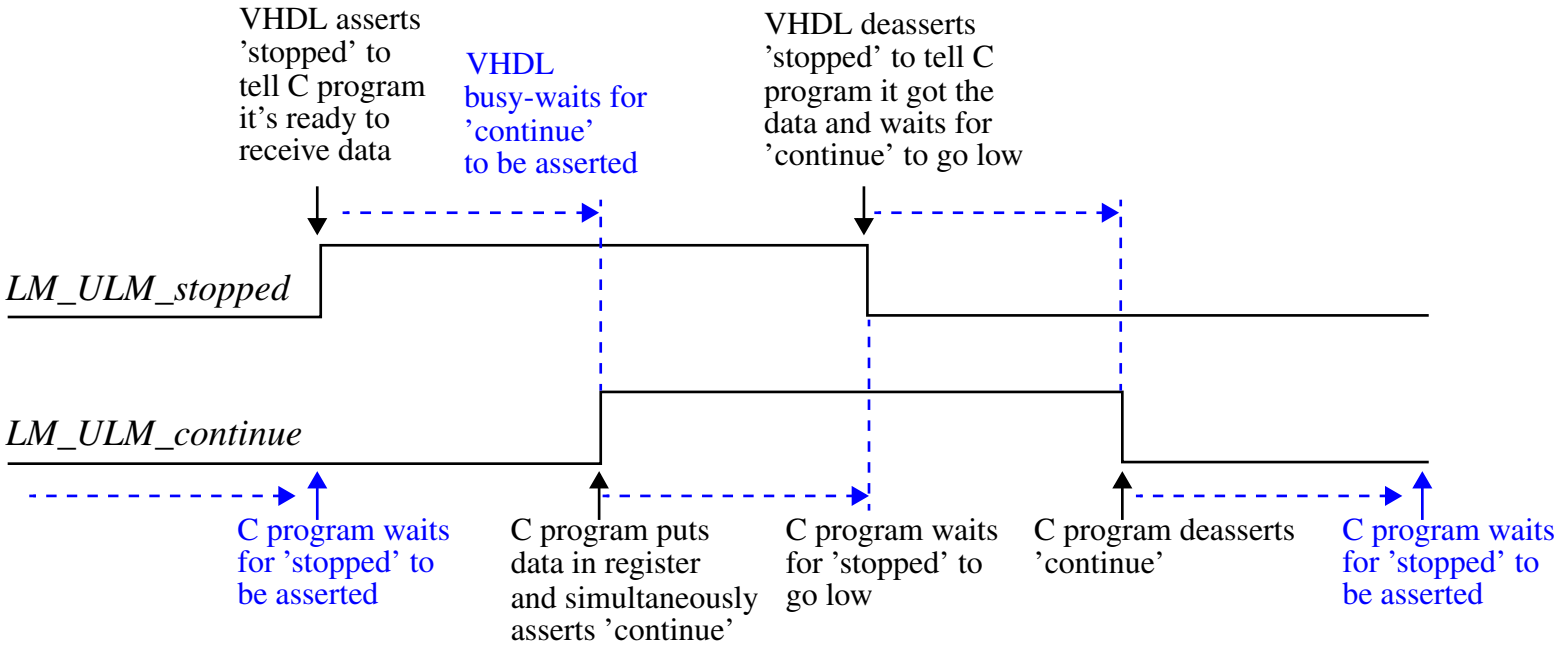
Therefore, we need a reliable protocol to allow transfers between PL and PS

HandShake Synchronization for Secure Memory Access Controller

Synchronization between our VHDL controller (PL) and the C program (PS) is done using a 2-way handshake, which makes use of two signals

LM_ULM_stopped and *LM_ULM_continue*

The following defines the protocol for data transfers **from** the C program **to** BRAM:



Data transfer from BRAM to C program is similar except for direction of data flow

This protocol ensures a reliable communication channel between the PS and PL side

Low Level Algorithm for Secure Memory Access Controller

We are now ready to describe the FSM at a lower level of detail

Some parts of the following pseudo-code may seem irrelevant, but in fact will make the algorithm more versatile

For example, we provide a *done* flag that the C program will assert to allow it to terminate the memory operations

In fact, *done* will allow the C program to terminate before doing any reads or writes whatsoever!

In our version, the C program starts the secure memory access controller (SMAC) FSM, but other usage scenarios may have other state machines start SMAC

In this case, the C program may need to inform SMAC that it has no read or write requests for the memory at that point in time

The C program operations in the following are designed to provide context, and are not part of the FSM

Algorithm for Secure Memory Access Controller

```
1) C program checks 'ready', sets 'load_unload' flag and
issues 'start' signal
2) C program waits for 'stopped'
3) idle: SMAC waits in idle for 'start'
   if ( start = '1' )
      Store 'base address' and 'upper limit' registers
      Check 'load_unload', if 0, Goto load_mem
      Check 'load_unload', if 1, Goto unload_mem
4) load_mem: Process write requests from C program
   Assert 'stopped' signal
   If 'done' is 0
      Check 'continue', if asserted, update BRAM
         Assert 'PNL_BRAM_we'
         Assign 'PNL_BRAM_din' GPIO data
         Goto wait_load_unload
   else
      Goto wait_done
```

Algorithm for Secure Memory Access Controller

```
5) unload_mem: Process read requests from C program
   Drive GPIO register with 'PNL_BRAM_dout' data
   Assert 'stopped'
   If 'done' is 0
      Check 'continue', if asserted, C program has data
      Goto wait_load_unload
   else
      Goto wait_done
6) wait_load_unload: Finish handshake and update addr
   Check 'continue', if 0
      If 'done', if 1
         Goto wait_done
      Elsif 'base addr' = 'upper_limit'
         Goto idle
      Else
         Inc 'base addr'
         Goto load_mem if 'load_unload' is 0
         Goto unload_mem if 'load_unload' is 1
```

Algorithm for Secure Memory Access Controller

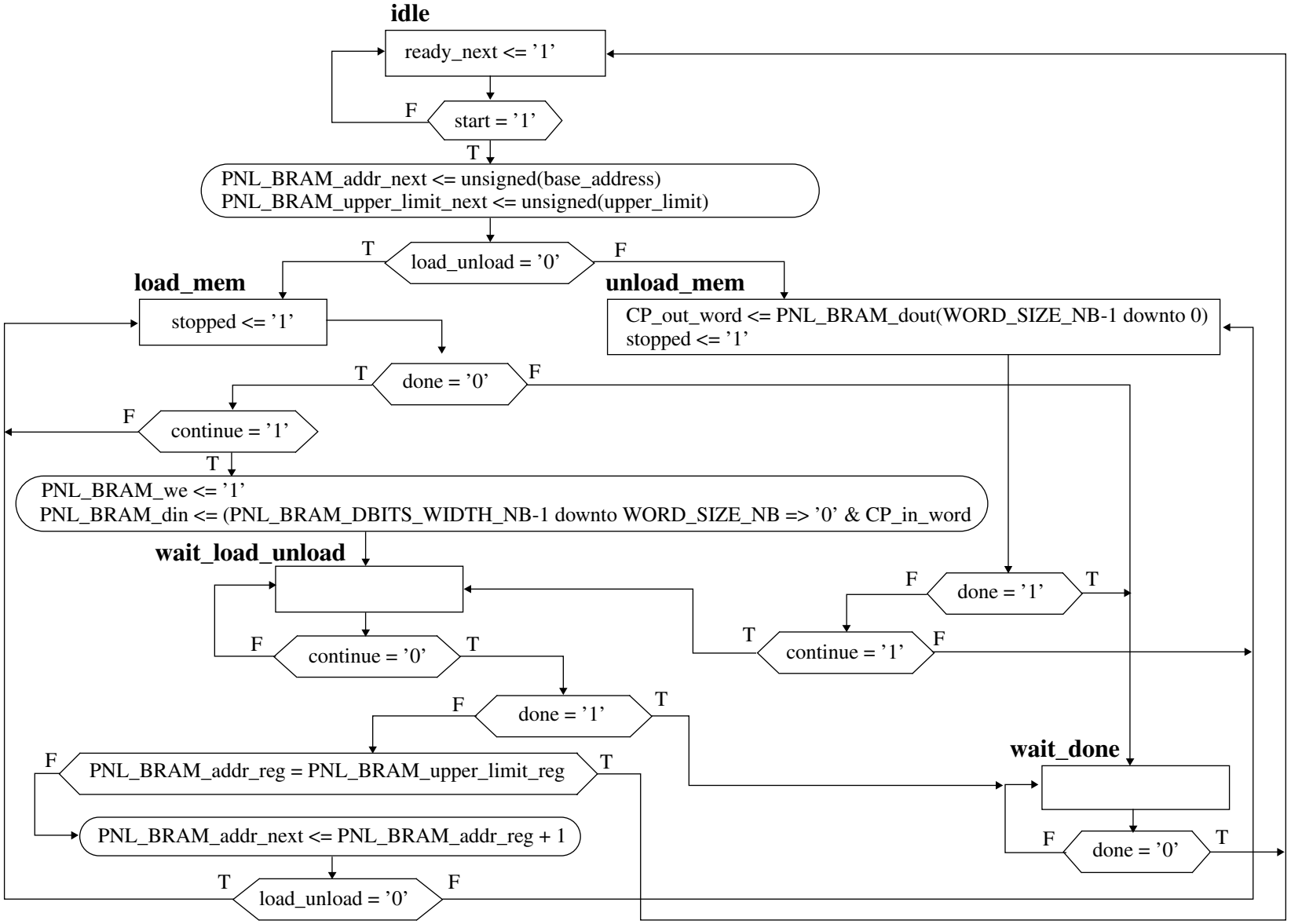
```
7) wait_done: Wait for C program to deassert 'done'
   Check 'done', if 0
      Goto idle
```

I was able to structure the pseudo-code directly into a form compatible with an FSM

As you can see, the lower level of abstraction has much more detail, with a 'goto-like' structure to implementing the state transition diagram

It also reveals elements of control in hardware design not found in software design, e.g., the 'write enable' (PNL_BRAM_we) associated with the memory

ASM For LoadUnloadMem.vhd



VHDL for DataTypes_pkg.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

package DataTypes_pkg is
   constant PN_NUMBITS_NB: integer := 12;
   constant PN_PRECISION_NB: integer := 4;
   constant PN_SIZE_LB: integer := 4;
   constant PN_SIZE_NB: integer := PN_NUMBITS_NB + PN_PRECISION_NB;

   constant BYTE_SIZE_LB: integer := 3;
   constant BYTE_SIZE_NB: integer := 8;

   constant WORD_SIZE_LB: integer := 4;
   constant WORD_SIZE_NB: integer := 16;

   constant PNL_BRAM_ADDR_SIZE_NB: integer := 13;
   constant PNL_BRAM_DBITS_WIDTH_LB: integer := PN_SIZE_LB;
   constant PNL_BRAM_DBITS_WIDTH_NB: integer := PN_SIZE_NB;
   constant PNL_BRAM_NUM_WORDS_NB: integer := 2**PNL_BRAM_ADDR_SIZE_NB;
   constant LARGEST_POS_VAL: integer := 16383;
   constant LARGEST_NEG_VAL: integer := -16383;
   constant PN_BRAM_BASE: integer := PNL_BRAM_NUM_WORDS_NB/2;
   constant PN_UPPER_LIMIT: integer := PNL_BRAM_NUM_WORDS_NB;
   constant DIST_BRAM_OFFSET: integer := (PNL_BRAM_NUM_WORDS_NB/2)/2;
end DataTypes_pkg;
```

VHDL for Top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

library work;
use work.DataTypes_pkg.all;

entity Top is
   port (
      Clk: in std_logic;
      PS_RESET_N: in std_logic;
      GPIO_Ins: in std_logic_vector(31 downto 0);
      GPIO_Outs: out std_logic_vector(31 downto 0);
      PNL_BRAM_addr: out std_logic_vector (PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
      PNL_BRAM_din: out std_logic_vector (PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_dout: in std_logic_vector (PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_we: out std_logic_vector (0 to 0);
      DEBUG_IN: in std_logic;
      DEBUG_OUT: out std_logic
      );
end Top;
```

VHDL for Top.vhd

```
architecture beh of Top is

-- GPIO INPUT BIT ASSIGNMENTS
   constant IN_CP_RESET: integer := 31;
   constant IN_CP_START: integer := 30;
   constant IN_CP_LM_ULM_LOAD_UNLOAD: integer := 26;
   constant IN_CP_LM_ULM_DONE: integer := 25;
   constant IN_CP_HANDSHAKE: integer := 24;

-- GPIO OUTPUT BIT ASSIGNMENTS
   constant OUT_SM_READY: integer := 31;
   constant OUT_SM_HANDSHAKE: integer := 28;

-- Signal declarations
   signal RESET: std_logic;

   signal LM_ULM_start, LM_ULM_ready: std_logic;
   signal LM_ULM_stopped, LM_ULM_continue: std_logic;
   signal LM_ULM_done: std_logic;
   signal LM_ULM_base_address: std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
   signal LM_ULM_upper_limit: std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
   signal LM_ULM_load_unload: std_logic;

   signal DataIn: std_logic_vector(WORD_SIZE_NB-1 downto 0);
   signal DataOut: std_logic_vector(WORD_SIZE_NB-1 downto 0);
```

VHDL for Top.vhd

```
    begin
    DEBUG_OUT <= LM_ULM_ready;
    RESET <= GPIO_Ins(IN_CP_RESET) or not PS_RESET_N;

-- =====================
-- INPUT control and status signals
    LM_ULM_start <= GPIO_Ins(IN_CP_START);
    LM_ULM_load_unload <= GPIO_Ins(IN_CP_LM_ULM_LOAD_UNLOAD);
    LM_ULM_done <= GPIO_Ins(IN_CP_LM_ULM_DONE);
    LM_ULM_continue <= GPIO_Ins(IN_CP_HANDSHAKE);
    DataIn <= GPIO_Ins(WORD_SIZE_NB-1 downto 0);

-- =====================
-- OUTPUT control and status signals
    GPIO_Outs(OUT_SM_READY) <= LM_ULM_ready;
    GPIO_Outs(OUT_SM_HANDSHAKE) <= LM_ULM_stopped;
    GPIO_Outs(WORD_SIZE_NB-1 downto 0) <= DataOut;

-- =====================
-- Setup memory base and upper_limit
    LM_ULM_base_address <=
       std_logic_vector(to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB));
    LM_ULM_upper_limit <=
       std_logic_vector(to_unsigned(PNL_BRAM_NUM_WORDS_NB -1,
          PNL_BRAM_ADDR_SIZE_NB));
```

VHDL for Top.vhd

```
-- Secure BRAM access control module
   LoadUnLoadMemMod: entity work.LoadUnLoadMem(beh)
      port map(Clk=>Clk, RESET=>RESET, start=>LM_ULM_start, ready=>LM_ULM_ready,
         load_unload=>LM_ULM_load_unload, stopped=>LM_ULM_stopped,
         continue=>LM_ULM_continue, done=>LM_ULM_done,
         base_address=>LM_ULM_base_address, upper_limit=>LM_ULM_upper_limit,
         CP_in_word=>DataIn, CP_out_word=>DataOut,
         PNL_BRAM_addr=>PNL_BRAM_addr, PNL_BRAM_din=>PNL_BRAM_din,
         PNL_BRAM_dout=>PNL_BRAM_dout, PNL_BRAM_we=>PNL_BRAM_we);
end beh;
```

VHDL for LoadUnloadMem.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;
library work;
use work.DataTypes_pkg.all;

entity LoadUnLoadMem is
   port(
      Clk: in std_logic;
      RESET: in std_logic;
      start: in std_logic;
      ready: out std_logic;
      load_unload: in std_logic;
      stopped: out std_logic;
      continue: in std_logic;
      done: in std_logic;
      base_address: in std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
      upper_limit: in std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
      CP_in_word: in std_logic_vector(WORD_SIZE_NB-1 downto 0);
      CP_out_word: out std_logic_vector(WORD_SIZE_NB-1 downto 0);
      PNL_BRAM_addr: out std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
      PNL_BRAM_din: out std_logic_vector(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_dout: in std_logic_vector(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_we: out std_logic_vector(0 to 0)
      );
end LoadUnLoadMem;
```

VHDL for LoadUnloadMem.vhd

```
architecture beh of LoadUnLoadMem is
   type state_type is (idle, load_mem, unload_mem, wait_load_unload, wait_done);
   signal state_reg, state_next: state_type;

   signal ready_reg, ready_next: std_logic;

   signal PNL_BRAM_addr_reg, PNL_BRAM_addr_next:
      unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
   signal PNL_BRAM_upper_limit_reg, PNL_BRAM_upper_limit_next:
      unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);

   begin
   process(Clk, RESET)
      begin
      if ( RESET = '1' ) then
         state_reg <= idle;
         ready_reg <= '1';
         PNL_BRAM_addr_reg <= (others=>'0');
         PNL_BRAM_upper_limit_reg <= (others=>'0');
      elsif ( Clk'event and Clk = '1' ) then
         state_reg <= state_next;
         ready_reg <= ready_next;
         PNL_BRAM_addr_reg <= PNL_BRAM_addr_next;
         PNL_BRAM_upper_limit_reg <= PNL_BRAM_upper_limit_next;
      end if;
   end process;
```

VHDL for LoadUnloadMem.vhd

```
process (state_reg, start, ready_reg, load_unload, PNL_BRAM_addr_reg,
   PNL_BRAM_upper_limit_reg, PNL_BRAM_dout, CP_in_word, continue, base_address,
   upper_limit, done)
   begin
   state_next <= state_reg;
   ready_next <= ready_reg;

   PNL_BRAM_addr_next <= PNL_BRAM_addr_reg;
   PNL_BRAM_upper_limit_next <= PNL_BRAM_upper_limit_reg;

   PNL_BRAM_we <= "0";
   PNL_BRAM_din <= (others=>'0');
   CP_out_word <= (others=>'0');

   stopped <= '0';
```

VHDL for LoadUnloadMem.vhd

```
        case state_reg is

    -- =====================
            when idle =>
               ready_next <= '1';
               if ( start = '1' ) then
                  ready_next <= '0';
                  PNL_BRAM_addr_next <= unsigned(base_address);
                  PNL_BRAM_upper_limit_next <= unsigned(upper_limit);

                  if ( load_unload = '0' ) then
                     state_next <= load_mem;
                  else
                     state_next <= unload_mem;
                  end if;
               end if;
```

VHDL for LoadUnloadMem.vhd

```
-- =====================
          when load_mem =>
             stopped <= '1';
             if ( done = '0' ) then
                if ( continue = '1' ) then
                   PNL_BRAM_we <= "1";
                   PNL_BRAM_din <= (PNL_BRAM_DBITS_WIDTH_NB-1 downto WORD_SIZE_NB =>
                     '0') & CP_in_word;
                   state_next <= wait_load_unload;
                end if;
             else
                state_next <= wait_done;
             end if;

-- =====================
          when unload_mem =>
             CP_out_word <= PNL_BRAM_dout(WORD_SIZE_NB-1 downto 0);
             stopped <= '1';
             if ( continue = '1' ) then
                state_next <= wait_load_unload;
             end if;

             if ( done = '1' ) then
                state_next <= wait_done;
             end if;
```

VHDL for LoadUnloadMem.vhd

```
-- =====================
          when wait_load_unload =>
             if ( continue = '0' ) then
                if ( done = '1' ) then
                   state_next <= wait_done;
                elsif ( PNL_BRAM_addr_reg = PNL_BRAM_upper_limit_reg ) then
                   state_next <= idle;
                else
                   PNL_BRAM_addr_next <= PNL_BRAM_addr_reg + 1;
                   if ( load_unload = '0' ) then
                      state_next <= load_mem;
                   else
                      state_next <= unload_mem;
                   end if;
                end if;
             end if;

-- =====================
          when wait_done =>
             if ( done = '0' ) then
                state_next <= idle;
             end if;
       end case;
    end process;
    PNL_BRAM_addr <= std_logic_vector(PNL_BRAM_addr_next);
    ready <= ready_reg;
end beh;
```

FSMDs

VHDL Essentials covers FSMD design principles

Here, we consider an interesting example that builds on the *secure memory access controller* (SMAC) discussed previously

The SMAC functionality allows C programs to load and unload an on-chip BRAM

We now add a module that performs an operation on the data stored in the BRAM

In particular, we will construct a **histogram** and compute the *mean* and *range* of the data values

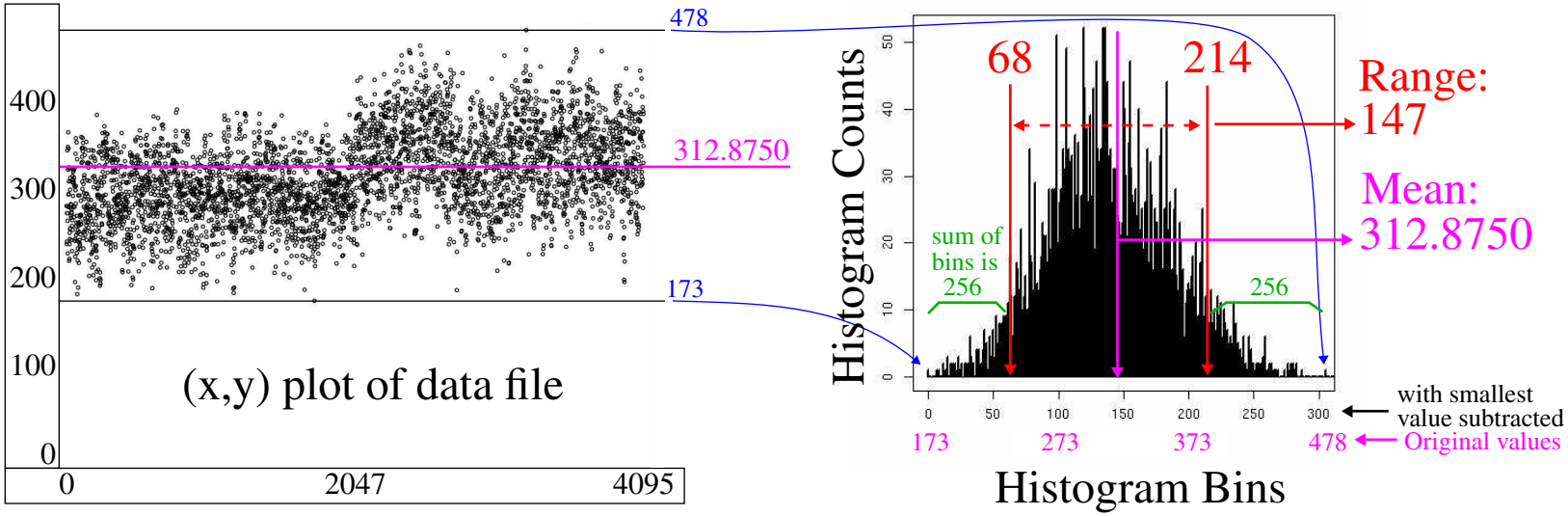
We use a C program as the starting point with this example

Interestingly, we can run both during the demo as we will see, and compare performances

Let's first consider the overall goals of our FSMD called **HISTO**

Histogram concepts

The data file that I supplied in the lab has 4096 fixed-point values



Our objective is to create a set of bins that contain counts, where each count is the number of points in the data file that possess a particular (integer) value

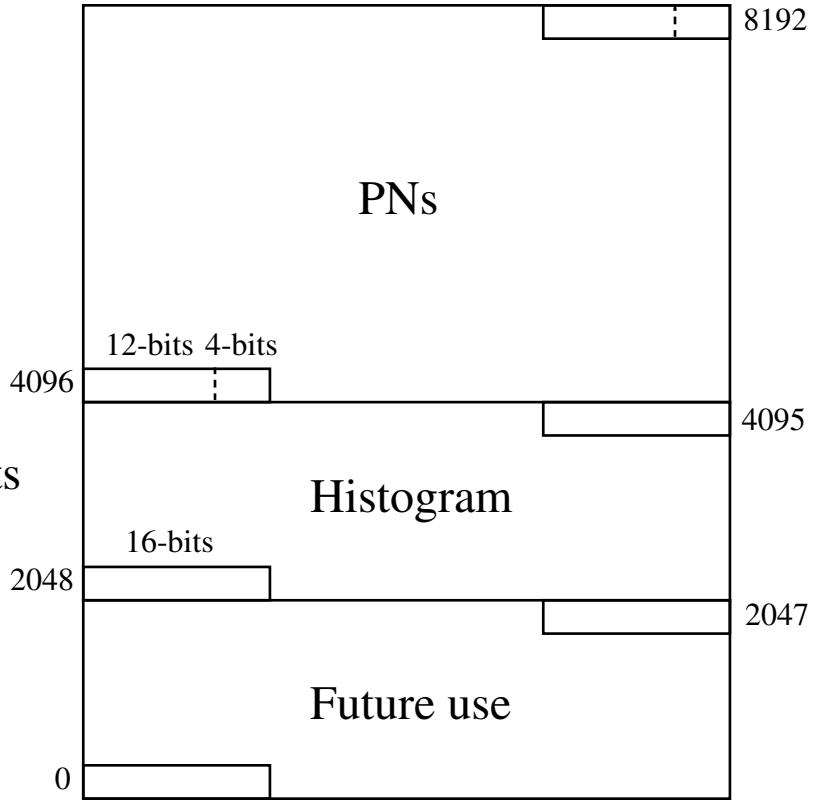
For example, there is only 1 point with value 173, so the histogram bar height is 1, while points near the mean occur as many as 50 times

The ***Range*** is computed between 6.25% and 93.75% points in the histogram, which are the points at which the sums of the bars in the tails are equal to 256

The ***Range*** considers on the integer portion while the ***Mean*** considers full precision

HISTO BRAM Organization

LoadUnloadMem loads the data file into upper half of memory



16-bit values allow counts upto 2^16 = 65535

We will use the address range between 2048 to 4097 for the histogram

The address range defines the *maximum range* of values present in the data file, i.e., the integer portion of the data values is used as the **address** into Histogram memory

HISTO C code

```
-- In the following, the array vals stores 4096 values from the data file,
-- LV_bound is 256, HV_bound is 3840 DIST_range is 2048 and precision_scaler is 16
-- software_histo stores the histogram

void ComputeHisto(int num_vals, short *vals,
   short LV_bound, short HV_bound,
   short DIST_range, short precision_scaler,
   short *software_histo)
   {
   short smallest_val, dist_cnt_sum, temp_val, range;
   int PN_num, bin_num, HISTO_ERR, dist_mean_sum;
   short LV_addr, HV_addr, LV_set, HV_set;

   HISTO_ERR = 0;
   dist_mean_sum = 0;

   for ( bin_num = 0; bin_num < DIST_range; bin_num++ )
      software_histo[bin_num] = 0;

   for ( PN_num = 0; PN_num < num_vals; PN_num++ )
      if ( PN_num == 0 )
         smallest_val = vals[PN_num];
      else if ( smallest_val > vals[PN_num] )
         smallest_val = vals[PN_num];
   smallest_val /= precision_scaler;
```

HISTO C code

```
for ( PN_num = 0; PN_num < num_vals; PN_num++ )
   {
   dist_mean_sum += (int)vals[PN_num];
   temp_val = vals[PN_num]/precision_scaler -
      smallest_val;
   if ( temp_val >= DIST_range )
      HISTO_ERR = 1;
   software_histo[temp_val]++;
   }

LV_addr = 0; HV_addr = 0;
LV_set = 0; HV_set = 0;
dist_cnt_sum = 0;
for ( bin_num = 0; bin_num < DIST_range; bin_num++ )
   {
   dist_cnt_sum += software_histo[bin_num];
   if ( LV_set == 0 && dist_cnt_sum >= LV_bound )
      {
      LV_addr = bin_num;
      LV_set = 1;
      }
   if ( dist_cnt_sum <= HV_bound )
      {
      HV_addr = bin_num;
      HV_set = 1;
      }
   }
range = HV_addr - LV_addr + 1;
```

HISTO C code

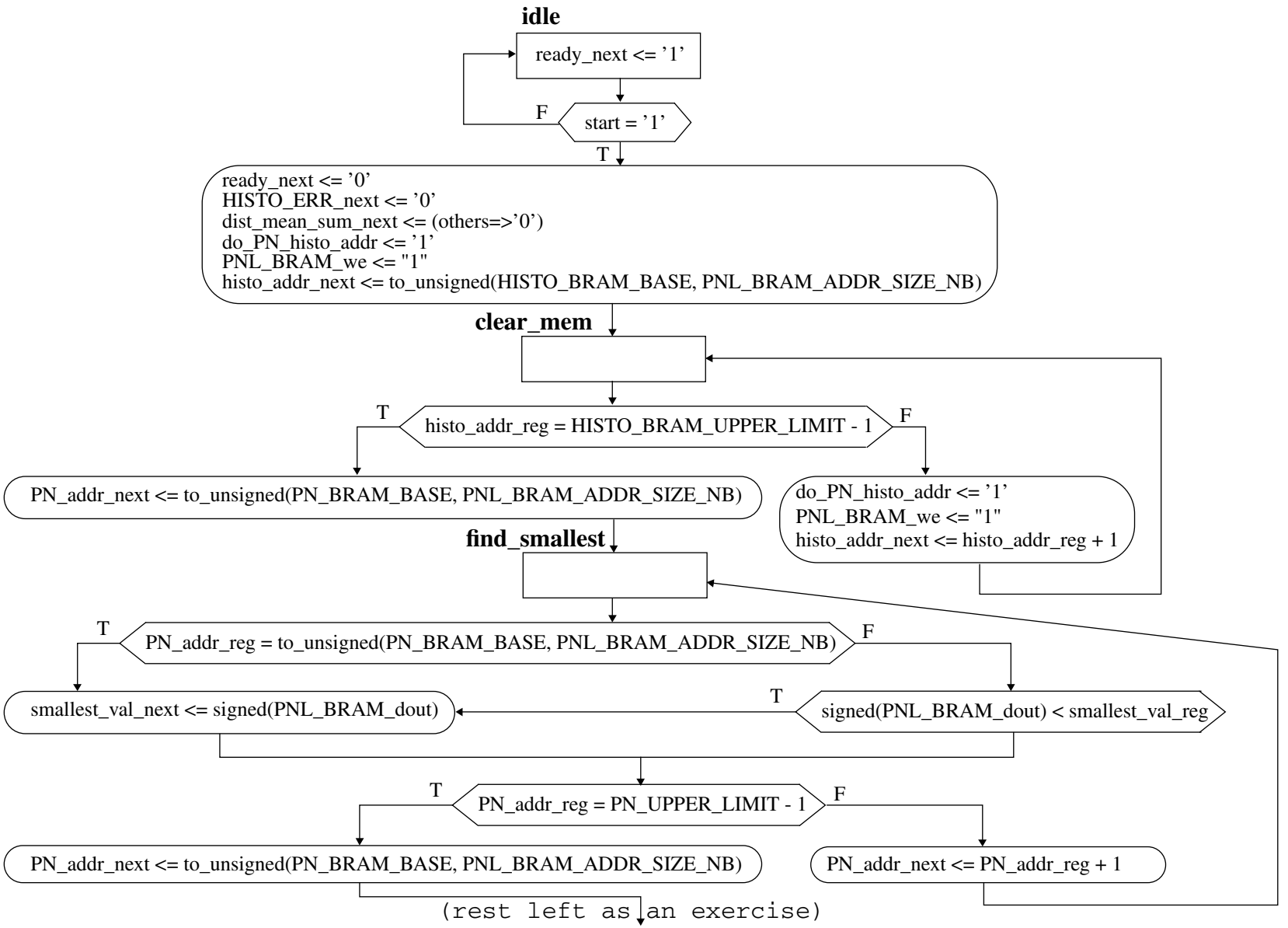
```
if ( LV_set == 0 || HV_set == 0 )
   HISTO_ERR = 1;

if ( HISTO_ERR == 1 )
   printf("ERROR: ComputeHisto(): Histo error!\n");

printf("Software Computed Stats: Smallest Val %d LV_addr %d  HV_addr %d
   Mean %.4f  Range %d\n", smallest_val, LV_addr, HV_addr,
   (float)(dist_mean_sum/num_vals)/precision_scaler,(int)range);

return;
}
```

HISTO ASMD


idle
ready_next <= '1'
F
start = '1'
T
ready_next <= '0'
HISTO_ERR_next <= '0'
dist_mean_sum_next <= (others=>'0')
do_PN_histo_addr <= '1'
PNL_BRAM_we <= "1"
histo_addr_next <= to_unsigned(HISTO_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB)
clear_mem
T
histo_addr_reg = HISTO_BRAM_UPPER_LIMIT - 1
F
PN_addr_next <= to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB)
do_PN_histo_addr <= '1'
PNL_BRAM_we <= "1"
histo_addr_next <= histo_addr_reg + 1
find_smallest
T
PN_addr_reg = to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB)
F
smallest_val_next <= signed(PNL_BRAM_dout)
T
signed(PNL_BRAM_dout) < smallest_val_reg
T
PN_addr_reg = PN_UPPER_LIMIT - 1
F
PN_addr_next <= to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB)
PN_addr_next <= PN_addr_reg + 1
(rest left as an exercise)

HISTO VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

library work;
use work.DataTypes_pkg.all;

entity Histo is
   port(
      Clk: in  std_logic;
      RESET: in std_logic;
      start: in std_logic;
      ready: out std_logic;
      HISTO_ERR: out std_logic;
      PNL_BRAM_addr: out std_logic_vector(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
      PNL_BRAM_din: out std_logic_vector(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_dout: in std_logic_vector(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
      PNL_BRAM_we: out std_logic_vector(0 to 0)
      );
end Histo;

architecture beh of Histo is
   type state_type is (idle, clear_mem, find_smallest, compute_addr, inc_cell,
      get_next_PN, init_dist, sweep_BRAM, check_histo_error, write_range);
   signal state_reg, state_next: state_type;

   signal ready_reg, ready_next: std_logic;

   signal PN_addr_reg, PN_addr_next: unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
```

HISTO VHDL

```
signal histo_addr_reg, histo_addr_next:
   unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);

signal do_PN_histo_addr: std_logic;

signal smallest_val_reg, smallest_val_next:
   signed(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);

signal shifted_dout: signed(PN_INTEGER_NB-1 downto 0);
signal shifted_smallest_val: signed(PN_INTEGER_NB-1 downto 0);

signal offset_addr: signed(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
signal histo_cell_addr: unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
signal LV_addr_reg, LV_addr_next: unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);
signal HV_addr_reg, HV_addr_next: unsigned(PNL_BRAM_ADDR_SIZE_NB-1 downto 0);

signal LV_set_reg, LV_set_next: std_logic;
signal HV_set_reg, HV_set_next: std_logic;

signal LV_bound, HV_bound: unsigned(NUM_PNS_NB downto 0);

signal dist_cnt_sum_reg, dist_cnt_sum_next: unsigned(NUM_PNS_NB downto 0);

signal dist_mean_sum_reg, dist_mean_sum_next:
   signed(NUM_PNS_NB+PN_SIZE_NB-1 downto 0);

signal dist_mean: std_logic_vector(PNL_BRAM_DBITS_WIDTH_NB-1 downto 0);
signal dist_range: std_logic_vector(HISTO_MAX_RANGE_NB-1 downto 0);
signal HISTO_ERR_reg, HISTO_ERR_next: std_logic;
```

HISTO VHDL

```
begin

dist_mean <= std_logic_vector(resize(dist_mean_sum_reg/(2**NUM_PNS_NB),
   PNL_BRAM_DBITS_WIDTH_NB));

dist_range <= std_logic_vector(resize(HV_addr_reg - LV_addr_reg + 1,
   HISTO_MAX_RANGE_NB));

process(Clk, RESET)
   begin
   if ( RESET = '1' ) then
      state_reg <= idle;
      ready_reg <= '1';
      PN_addr_reg <= (others => '0');
      histo_addr_reg <= (others => '0');
      smallest_val_reg <= (others => '0');
      LV_addr_reg <= (others => '0');
      HV_addr_reg <= (others => '0');
      LV_set_reg <= '0';
      HV_set_reg <= '0';
      dist_cnt_sum_reg <= (others => '0');
      dist_mean_sum_reg <= (others => '0');
      HISTO_ERR_reg <= '0';
   elsif ( Clk'event and Clk = '1' ) then
      state_reg <= state_next;
      ready_reg <= ready_next;
      PN_addr_reg <= PN_addr_next;
      histo_addr_reg <= histo_addr_next;
      smallest_val_reg <= smallest_val_next;
```

HISTO VHDL

```
            LV_addr_reg <= LV_addr_next;
            HV_addr_reg <= HV_addr_next;
            LV_set_reg <= LV_set_next;
            HV_set_reg <= HV_set_next;
            dist_cnt_sum_reg <= dist_cnt_sum_next;
            dist_mean_sum_reg <= dist_mean_sum_next;
            HISTO_ERR_reg <= HISTO_ERR_next;
         end if;
      end process;

      shifted_dout <= resize(signed(PNL_BRAM_dout)/16, PN_INTEGER_NB);
      shifted_smallest_val <= resize(smallest_val_reg/16, PN_INTEGER_NB);

      offset_addr <= resize(shifted_dout, PNL_BRAM_ADDR_SIZE_NB) -
         resize(shifted_smallest_val, PNL_BRAM_ADDR_SIZE_NB);

      histo_cell_addr <= unsigned(offset_addr) + to_unsigned(HISTO_BRAM_BASE,
         PNL_BRAM_ADDR_SIZE_NB);

      LV_bound <= to_unsigned(NUM_PNs, NUM_PNS_NB+1) srl HISTO_BOUND_PCT_SHIFT_NB;
      HV_bound <= to_unsigned(NUM_PNs, NUM_PNS_NB+1) - LV_bound;
```

HISTO VHDL

```
-- ================================================================================
-- Combo logic
-- ================================================================================

   process (state_reg, start, ready_reg, PN_addr_reg, histo_addr_reg,
      PNL_BRAM_dout, histo_cell_addr, LV_bound, HV_bound, LV_addr_reg,
      HV_addr_reg, LV_set_reg, HV_set_reg, dist_cnt_sum_reg,
      dist_cnt_sum_next, dist_mean_sum_reg, smallest_val_reg, dist_mean,
      dist_range, HISTO_ERR_reg)
      begin
      state_next <= state_reg;
      ready_next <= ready_reg;

      PN_addr_next <= PN_addr_reg;
      histo_addr_next <= histo_addr_reg;
      smallest_val_next <= smallest_val_reg;
      LV_addr_next <= LV_addr_reg;
      HV_addr_next <= HV_addr_reg;
      LV_set_next <= LV_set_reg;
      HV_set_next <= HV_set_reg;
      dist_cnt_sum_next <= dist_cnt_sum_reg;
      dist_mean_sum_next <= dist_mean_sum_reg;
      HISTO_ERR_next <= HISTO_ERR_reg;

      PNL_BRAM_din <= (others=>'0');
      PNL_BRAM_we <= "0";

      do_PN_histo_addr <= '0';
```

HISTO VHDL

```
            case state_reg is
               when idle =>
                  ready_next <= '1';

                  if ( start = '1' ) then
                     ready_next <= '0';

                     HISTO_ERR_next <= '0';

                     dist_mean_sum_next <= (others=>'0');

                     do_PN_histo_addr <= '1';

                     PNL_BRAM_we <= "1";
                     histo_addr_next <= to_unsigned(HISTO_BRAM_BASE,
                        PNL_BRAM_ADDR_SIZE_NB);
                     state_next <= clear_mem;
                  end if;

      -- =====================
               when clear_mem =>
                  if ( histo_addr_reg = HISTO_BRAM_UPPER_LIMIT - 1 ) then
                     PN_addr_next <= to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB);
                     state_next <= find_smallest;
                  else
                     do_PN_histo_addr <= '1';
                     PNL_BRAM_we <= "1";
                     histo_addr_next <= histo_addr_reg + 1;
                  end if;
```

HISTO VHDL

```
-- =====================
        when find_smallest =>
           if ( PN_addr_reg =
              to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB) ) then
              smallest_val_next <= signed(PNL_BRAM_dout);
           elsif ( signed(PNL_BRAM_dout) < smallest_val_reg ) then
              smallest_val_next <= signed(PNL_BRAM_dout);
           end if;

           if ( PN_addr_reg = PN_UPPER_LIMIT - 1 ) then
              PN_addr_next <= to_unsigned(PN_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB);
              state_next <= compute_addr;
           else
              PN_addr_next <= PN_addr_reg + 1;
           end if;

-- =====================
        when compute_addr =>
           do_PN_histo_addr <= '1';
           histo_addr_next <= histo_cell_addr;

           if ( histo_cell_addr > HISTO_BRAM_UPPER_LIMIT - 1 ) then
              HISTO_ERR_next <= '1';
           end if;

           dist_mean_sum_next <= dist_mean_sum_reg + signed(PNL_BRAM_dout);

           state_next <= inc_cell;
```

HISTO VHDL

```
-- =====================
        when inc_cell =>
           do_PN_histo_addr <= '1';
           PNL_BRAM_we <= "1";
           PNL_BRAM_din <= std_logic_vector(unsigned(PNL_BRAM_dout) + 1);
           state_next <= get_next_PN;

-- =====================
        when get_next_PN =>
           if ( PN_addr_reg = PN_UPPER_LIMIT - 1 ) then
              state_next <= init_dist;
           else
              PN_addr_next <= PN_addr_reg + 1;
              state_next <= compute_addr;
           end if;

-- =====================
        when init_dist =>
           do_PN_histo_addr <= '1';
           histo_addr_next <= to_unsigned(HISTO_BRAM_BASE, PNL_BRAM_ADDR_SIZE_NB);

           LV_addr_next <= (others => '0');
           HV_addr_next <= (others => '0');

           LV_set_next <= '0';
           HV_set_next <= '0';
           dist_cnt_sum_next <= (others => '0');

           state_next <= sweep_BRAM;
```

HISTO VHDL

```
-- ====================
        when sweep_BRAM =>
           do_PN_histo_addr <= '1';
           dist_cnt_sum_next <= dist_cnt_sum_reg +
              resize(unsigned(PNL_BRAM_dout), NUM_PNS_NB+1);

           if ( LV_set_reg = '0' and dist_cnt_sum_next >= LV_bound ) then
              LV_addr_next <= histo_addr_reg;
              LV_set_next <= '1';
           end if;

           if ( dist_cnt_sum_next <= HV_bound ) then
              HV_addr_next <= histo_addr_reg;
              HV_set_next <= '1';
           end if;

           if ( histo_addr_reg = HISTO_BRAM_UPPER_LIMIT - 1 ) then
              state_next <= check_histo_error;
           else
              histo_addr_next <= histo_addr_reg + 1;
           end if;
```

HISTO VHDL

```
-- =====================
        when check_histo_error =>
           if ( LV_set_reg = '0' or HV_set_reg = '0' ) then
              HISTO_ERR_next <= '1';
              state_next <= idle;
           else
              do_PN_histo_addr <= '1';
              histo_addr_next <= to_unsigned(HISTO_BRAM_UPPER_LIMIT - 2,
                 PNL_BRAM_ADDR_SIZE_NB);
              PNL_BRAM_we <= "1";
              PNL_BRAM_din <= dist_mean;
              state_next <= write_range;
           end if;

-- =====================
        when write_range =>
           do_PN_histo_addr <= '1';
           histo_addr_next <= to_unsigned(HISTO_BRAM_UPPER_LIMIT - 1,
              PNL_BRAM_ADDR_SIZE_NB);
           PNL_BRAM_we <= "1";
           PNL_BRAM_din <= (PNL_BRAM_DBITS_WIDTH_NB-1 downto
              HISTO_MAX_RANGE_NB => '0') & dist_range;
           state_next <= idle;

      end case;
   end process;
```

HISTO VHDL

```
    with do_PN_histo_addr select
       PNL_BRAM_addr <= std_logic_vector(PN_addr_next) when '0',
                        std_logic_vector(histo_addr_next) when others;
    HISTO_ERR <= HISTO_ERR_reg;
    ready <= ready_reg;

end beh;
```