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 compli- cated 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

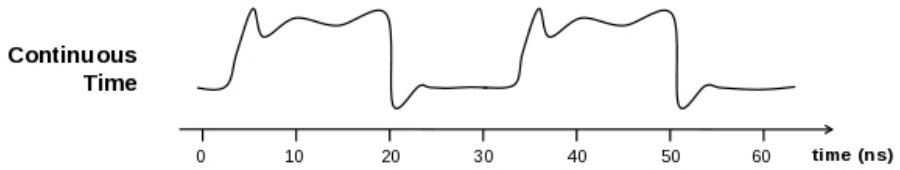
Abstraction refers to the ***level of detail*** that is available in a model

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

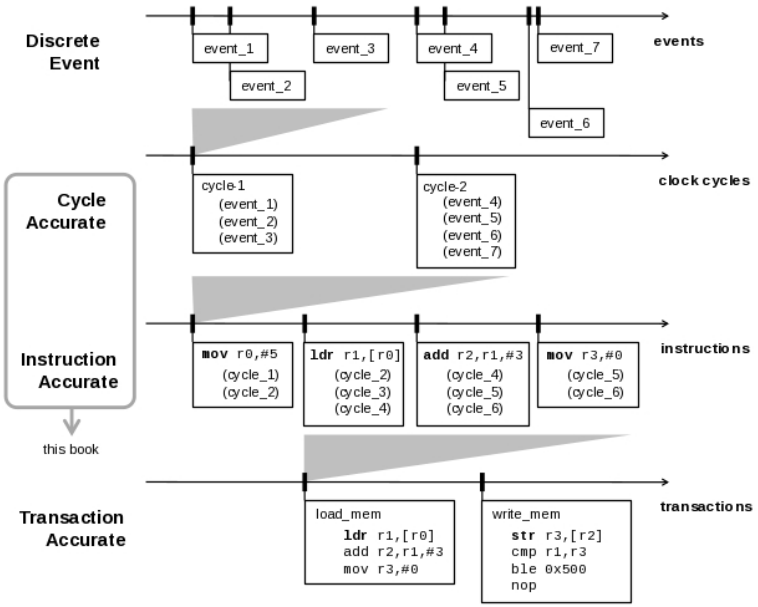
• **Continuous time** (lowest level):



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 opera- tions 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**

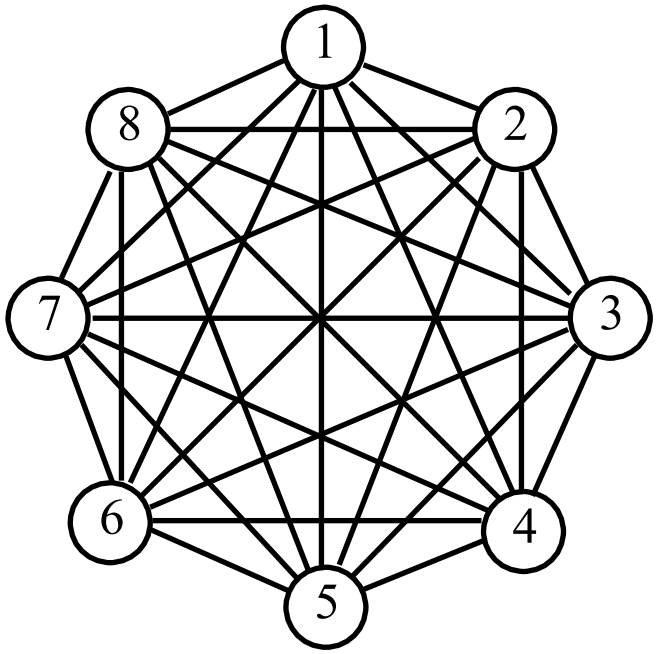
C programs are sequential, and so are typical instruction set architectures

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?

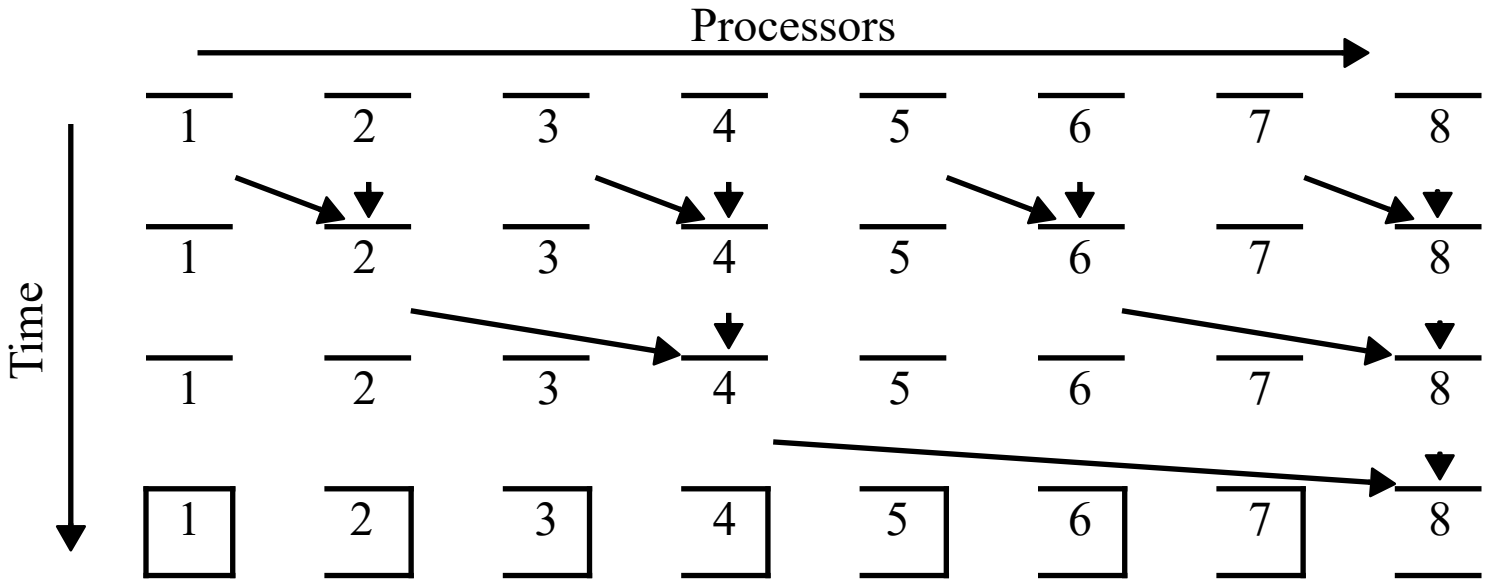
It's possible to write individual C programs for each node, but this is really not practical with 64K nodes!

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

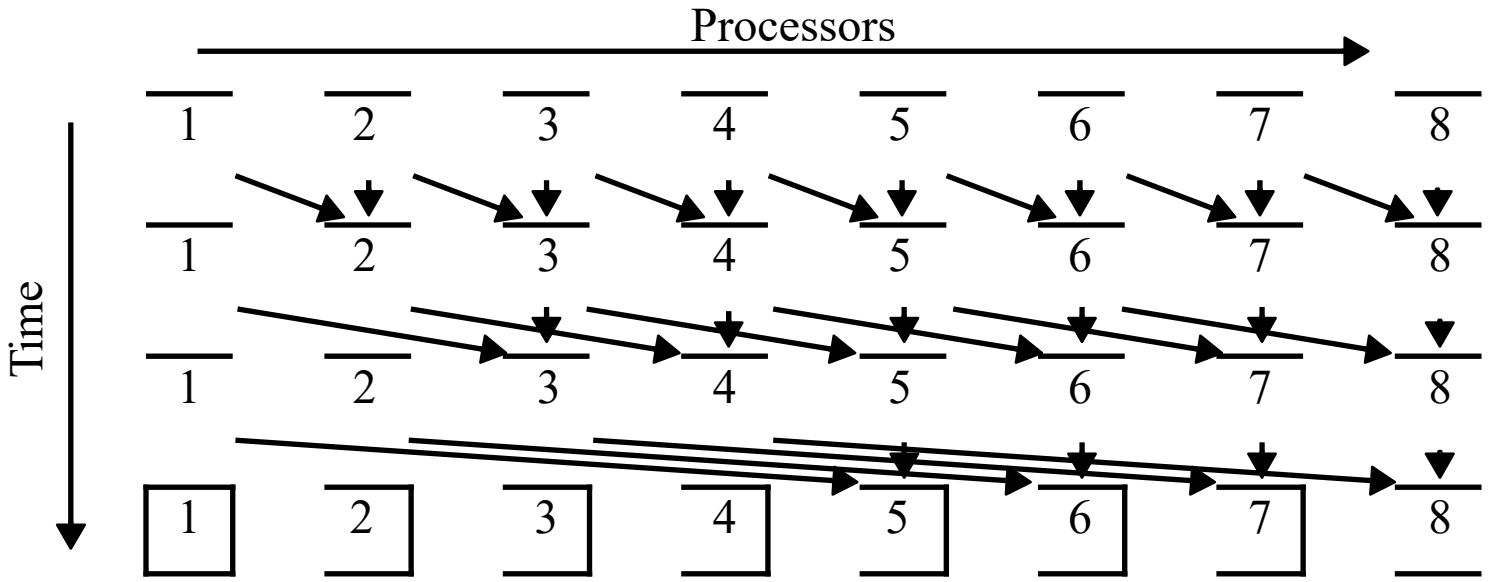
The same algorithm running on a sequential processor would take 7 steps

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 sequen- tial (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