Defining HW/SW Models (A Practical Introduction to HW/SW Codesign, P. Schau-mont)

The purpose of this discussion is to compare HW and SW and identify the differences and similarities between them.

Hardware

For this course, **hardware** means *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. The behavior of the circuit is **abstracted** as a set of operations performed on data as it is transferred between registers.
Hardware

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

A *single-thread sequential* program for software can be a 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 }
```

**Listing 1.2** ARM assembly example

```
.text

findmax:
  ldr r2, .L10
  ldr r3, [r0, #0]
  str r3, [r2, #0]
```
We will keep most discussions processor-independent, and will consider 32-bit architectures, e.g., ARM and Microblaze, and 8-bit archs, e.g., 8051, Picoblaze
Software
HW/SW codesign deals with low-level HW/SW interfaces

We will consider important SW implementation aspects such as
- The various sections of memory (global, stack, heap)
- The different kinds of memory (registers, caches, RAM and ROM)
- Techniques to control their use from within a high-level programming language
  such as C

The book emphasizes the relationship between C and assembly code
  We assume you are familiar with the concepts of assembly

Some optimization problems can only be solved at the level of assembly coding
  Most C compilers allow for the automatic generation of assembly

Modeling Abstraction Levels
Even given our restriction to single-clock synchronous HW and single-thread sequential SW, there are many different approaches to describe these models
Software

We will differentiate the abstraction levels based on their **time-granularity**
A small time-granularity typically means less work gets done per step

There are five abstraction levels

- **Continuous time** (lowest level):
  Operations described as continuous actions, e.g., charging and discharging voltages and currents can be expressed using differential equations

  Although useful to analyze the analog effects in deep-submicron technologies, too slow for HW/SW codesign

- **Discrete-event**:
  Activity lumped into discrete, possibly irregularly spaced, time steps called *events*

  Changing the inputs of a digital combinatorial circuit causes values to ripple through the circuit
  The speed of the change is limited by the propagation delay of the gates
Software

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

  Discrete-event simulation is very popular to model hardware at low abstraction level

  Model is simpler than continuous time but accurate enough to capture all relevant information such as glitches and clock cycle edges.

  Discrete-event simulation is also used to model higher abstraction-level systems, e.g., customer queues at a bank

- **Cycle-accurate**:

  All interesting events happen in single-clock synchronous systems at regularly-spaced intervals, i.e., the $clk$ edge

  A cycle-accurate model **does not** capture propagation delays or glitches
  
  Activities happen either immediately for combinatorial circuits or after an integral number of clock cycles for sequential circuits
Software

- **Cycle-accurate**: (cont.)
  
  Modern HW synthesis tools and flows has made cycle-accurate modeling level the "golden reference" for HW/SW system description.

- **Instruction-accurate**:
  
  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).

  The instruction-accurate modeling level 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 actual execution time (not trivial).

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

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

  In transaction-accurate modeling, the behavior is expressed in terms of the interactions (transactions) between the components of a system.

  For example, you can model a system with a disk drive and a user application. You simulate commands exchanged between the disk drive and the user application.

  In between two transactions, millions of instructions are lumped together and simulated as a single, atomic 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.
Fundamental Differences between 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.

Consider the differences:

- **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.

- **Temporal vs. spatial decomposition**

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

  The resources of hardware and software are **duals**.
Hardware and Software and Their Differences

- Temporal vs. spatial decomposition (cont.)
  
  When more resources are allocated in hardware, execution time is kept under tight control at the expense of circuit complexity & area

  When more software operations are added, execution time increases but \textit{circuit} complexity remains constant

  Hence, hardware engineers use \textit{spatial decomposition} by investing circuit area to maintain constant execution time

  And software engineers use \textit{temporal decomposition} by investing in execution time to maintain constant circuit area

  This simplification illustrates an important concept:
  
  HW engineers are challenged by \textbf{area constraints} while SW engineers are challenged by \textbf{time constraints}
Hardware and Software and Their Differences

- **Flexibility**
  
  Flexibility is the *ease* an application can be modified or adapted to map onto a target architecture

  **Software clearly excels** over hardware in its support for application flexibility

  Flexibility is easily implemented in software and is effectively ’free’

  In hardware, flexibility is *non-trivial*. It requires the **reuse** of circuit elements for different activities/functions in a design
  
  Flexibility is implemented in hardware via MUXs and additional control signals

- **Control processing vs. data processing**
  
  Control processing involves the use of decision constructs, e.g., control variables are used in *if-then-else* and *while-loops*
  
  Do A or B based on *condition*
Hardware and Software and Their Differences

• Control processing vs. data processing (cont.)
  Software is much better at complex control processing than hardware

In contrast, hardware is better at data processing
  For intense data crunching tasks, such as those associated with DSP pipelines, increasing parallelism is easy, e.g., add another multiplier

  Complex data processing for software requires additional instructions and longer program running times

• Modeling and Implementation
  In software, modeling and implementation are similar
    A C program is the model, its compilation is its implementation

  In hardware, models and implementations of a design are distinct
    A circuit is first described (modeled) using HDL or as a schematic
    In order to implement it, hardware synthesis is required -- somewhat analogous to compilation but the process is more complex
**Hardware and Software and Their Differences**

- Intellectual Property reuse
  
  IP-reuse: An expert designs a component of a larger circuit or program and *sells* that component individually

  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

  From this comparison, it is apparent that hardware and software are **duals**
  
  It is important for you to develop skills to transition from hardware concepts to software concepts and vise versa

  To master **system design**, you need to be an expert in both areas
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 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 requires a single processor, 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

An important driver for a software designer to use HW/SW codesign is that allows for the creation of a parallel implementation.

Assuming the underlying specification contains enough concurrency

Remember the famous 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 you can make the parallel portion run \textbf{arbitrarily fast}.

However, the sequential part cannot leverage this parallelism, so the max speedup is 3

The task of making your application take advantage of parallelism is \textbf{not obvious}.

C programs are sequential, and so are typical instruction set architectures.
Concurrency and Parallelism

Consider a simple addition application mapped onto 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

Today, there is a trend to build such architectures on a single chip (SoC)

How hard is it to write programs for this machine?
It’s possible to write individual C programs for each node, but not practical with 64K nodes!
Concurrency and Parallelism

The authors of the Connection Machine, Hellis and Steele, show that it is possible to express algorithms in a concurrent fashion so that they map nicely to the 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 -- in this example, 3 time steps.

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 the 8-node CM for 3 times steps is 3*8 = 24 computation time-steps of which only 7 are being used.

A simple modification to the algorithm to compute all partial sums, i.e., the sum of the first two, three, four, etc. numbers, makes better use of compute power.

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)

Conclusion: you can make optimal use of the underlying hardware if you develop a concurrent specification

If you restrict yourself to a sequential specification, it will be much harder for you 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