Embedded Systems Design with Platform FPGAs

Chapter -5 Spatial Design

Dr. Bassam Jamil
Adopted and updated from
Ron Sass and Andrew G. Schmidt

Chapter 5 — Spatial Design

Outline of This Chapter

Spatial Design

- Single vs. Multiple threads:
  - Designs with **a single thread** of control: each instruction or operation is executed fully **prior to** execution of the next.
  - **Spatial Design**:
    - allows for **multiple** threads of control to be executed **simultaneously**.
    - In computer science terms, this is more generally known as **concurrency**.
- Example: Consider performing a search over a large array of numbers.
  - linear search: O(***n***)
  - binary search:
    - Single processor: O(log ***n***).
    - fully parallel search implementation: O(1)
- It may not always be feasible to implement such a large number of parallel operations,
  - it is worth considering how to design systems that are able to take advantage of the exceptional parallelism available within FPGAs.

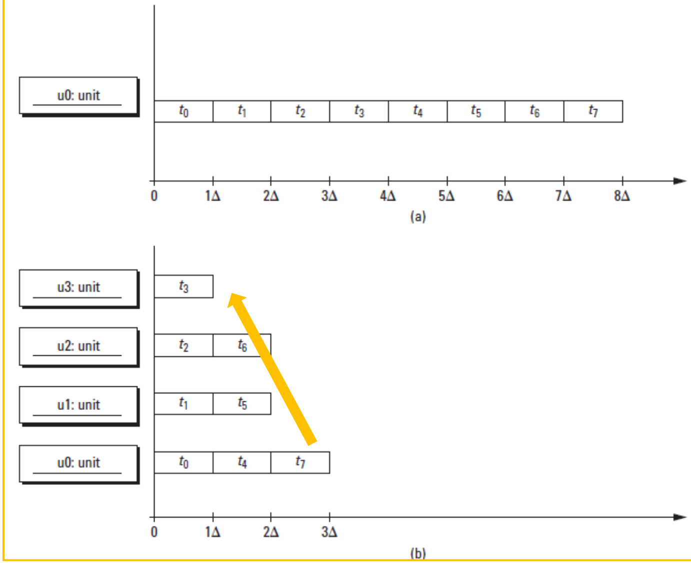
Outline of This Chapter

5.1 Principles of Parallelism

- Computation can be described a:
  - –**Sequential**: The same components are resused over time and the results are the accumulation of a **sequence** of operations.
  - –**Parallel**: Multiple instances of components are used and some operations may happen **concurrently**.
- Example (See Next Slide):
  - Consider a task T: that is composed of a set of eight subtasks
  - T= {t0, t1, . . . , t7}.
  - T is executed sequentially and with four unites.
  - **Sequentially** on a single component
    - task T takes 8△ to complete.
  - **Four units** only takes 3△ to complete task T.
  - Speedup=2.1 , but improved design needs 4× resources.

Example

- Speed up=2.1
- Second design requires 4× the resources.



"**Why not move t7 to unit 3 in time slot [1 △, 2 △]?** Then the application will complete 4× faster.
Answer: If the programmer who developed the application assumed that it would be executed sequentially then any parallel execution of T 's subtasks must respect this ordering.

So fundamental issue of this chapter: discover the parallelism in the sequential software reference design.

5.1.1. Granuality

- T(which is software reference design)is set of subtasks {ti} .
- Two ways to define the sub-tasks.
  - **Fine-grain** parallelism: subtasks ti are relatively small, maybe the equivalent of a **few instructions**.
  - **Coarse-grain** parallelism: relatively large subtasks, such as an **entire subroutine**.
- The following table summarizes the types and features of granularity based on how large the sub-tasks are.

5.1.1. Granularity

Level of Parallelism	T= {t1, t2, …..}	Sub-tasks: {ti}	Notes
Bit-level	**T** operates on scalar values	Bit fields: organize parallel **ti**'s to work on each field in parallel	Advantage: independent of the application Disadvantage: the number of bits in the scalar tends to limit the parallelism.
Instruction -level	Program	instructions	Advantage: increases parallelism of sequential code.
Iteration-level (data-level)	**T** operates on Vectors	Sequence of instruction on vectors	Example: inner product $c = \sum_i a_i \times b_i$
Thread-level	Parallelism explicitly in multiple threads	Each sub-task is a thread, which does not execute the same instructions	

5.1.2. Degree of Parallelism

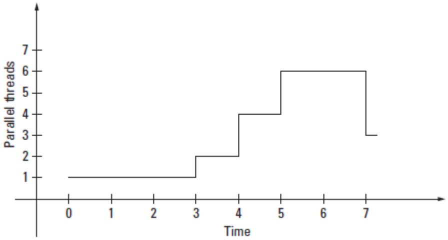
- Degree of parallelism (DOP) can refer to
  - The number of concurrent operations at a single moment of time,
  - A time varying function over the entire application, or
  - The average of a time-varying function over some task T, which may not include the entire application.

In the Figure:

- **Maximum DOP** is 6, which occurs between time 5–7.
- **Average DOP** is 3.
  - Average = Area/Time=21/7=3

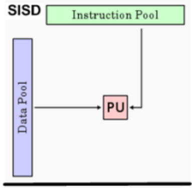
What this tells us is that:

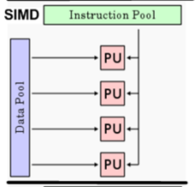
- if these are iteration-level subtasks, having more than six cores would be overkill.
- Likewise, if hardware resources were scarce, having four cores instead of six would achieve 87% of the performance with 66% of the resources.

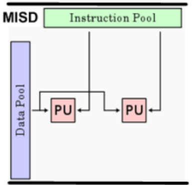


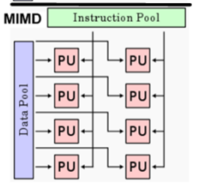
5.1.3. Spatial Organizations

- Taxonomy is based on instruction stream (single/multiple) and data stream (single/multiple).
- **Single Instruction Stream, Single Data Stream (SISD)**
  - Single processing unit consumes a single sequence (stream) of data.
  - Used in many general purpose computers.
- **Single Instruction Stream, Multiple Data Stream (SIMD)**
  - Uses simple Processing Elements (PEs) that are all performing instructions from a **single instruction** stream in lock step.
- **Multiple Instruction Stream, Single Data Stream (MISD)**
  - Breaks up a single stream of instructions (or subinstructions) into independent operations to be executed in parallel.
  - These independent streams of instructions are then assigned to separate processors, and a single **stream of data is passed from processor to processor** via explicit communication channels.
- **Multiple Instruction Stream, Multiple Data Stream (MIMD)**
  - Multiple independent processors, each with its own memory and own instructions.
  - The processors might communicate via some shared memory hierarchy or via explicit communication channels.



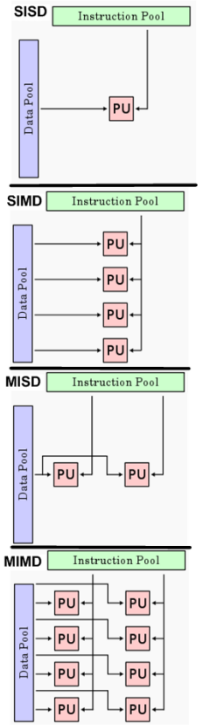






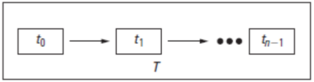
Spatial Organizations: Examples

- **SIMD** : used for special instruction sets added to commodity processors and Graphical Processor Units (GPU) used in video cards.
- Also, examples of **SIMD**
  – Vector processors, pipeline processors
  – Stream processors
  – Difference between vector and stream processor:
    - In vector processors, the programmer just had to follow a handful of guidelines and the **compiler** was primarily responsible for detecting and implementing vector instructions.
    - In stream processors, the **programmer** is usually tasked with explicitly identifying “kernels” in their code — each which has its own local memory and will ultimately be mapped to an independent processor.
- **MISD:**
  – “systolic architectures” because it resembled blood flowing along arteries and veins.
- **MIMD**
  – In MIMD, the hardware could execute separate instruction streams,
  – It is possible to replicate the same instruction stream on every processor. This programming model, called Single Program Multiple Data (SPMD) (or data parallel programming).
  – SPMD is productive way to write scalable parallel applications.

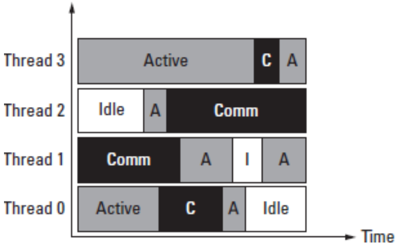


Vector and Pipelined Parallelism

- Assume:
  - Task T = $\{t_0, t_1, \ldots, t_{n-1}\}$
  - each task buffers its outputs with a register, clocks this core at $\triangle = \max_{t \in T} \{delay(t)\}$
  - a sequence of inputs **I** that produces a sequence of outputs $O = \langle o_0, o_1, \ldots o_{m-1} \rangle$
  - then it takes n cycles to produce a result
  - Throughput (outputs/cycle) = $\frac{m}{m+n}$
  - Assuming that n << m, then throughput approaches 1
- Embedded systems move and process large vectors, much like any computing system.
- By starting a new operation before the last one completes,
  - we are effectively overlapping consecutive operations of T. The time-space diagram shown in the Figure illustrates this for four subtasks.
  - Assuming that n << m, then the degree of parallelism is the number of threads



Pipeline parallelism.



Control Parallelism (Irregular or ad hoc parallelism)

- Examines a single stream of instructions to find operations that can be performed across multiple components in parallel.
- Control parallelism can be utilized at different granularity.
  - We will examine statement and operation granularity.
- Consider the following program that computes the roots of the polynomial $ax^2 + bx + c$ with the quadratic formula:

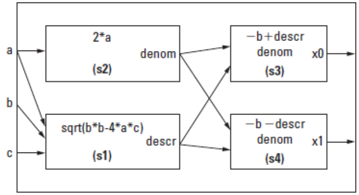
$$x_0, x_1 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- The sequential codewouldbe:

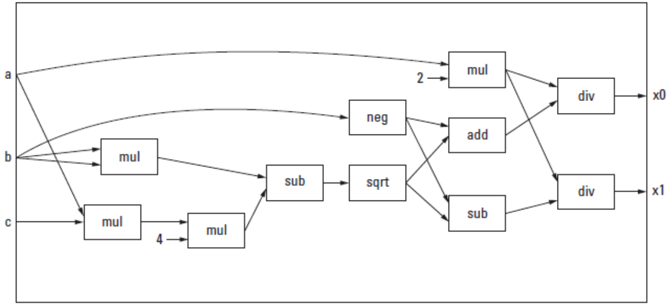
```
(s1) descr = sqrt(b*b-4*a*c)
(s2) denom = 2*a
(s3) x0 = (-b+descr)/denom
(s4) x1 = (-b-descr)/denom
```

Control Parallelism: Example

- Parallelism at **Statement Granularity**
  - instructions s1 and s2 can operate in parallel,
  - but s3 and s4 cannot begin until the first two finish.
  - However, once s1 and s2 have finished, then the last two can be performed separately.
  - the solution would have a constant degree of parallelism of two



- Parallelism at **Operation Granularity**



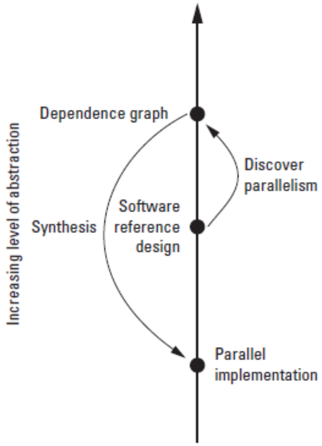
Data (Regular) Parallelism

- Data parallelism focuses on organizing the parallel threads of control around a regular data structure, such as an array, and loops.
- If each iteration of a loop operating on an element of an array is independent, then it is **easy to divide up the array among the available resources**.
- In the event that the design is migrated to a **larger chip** with more resources, the array can be again divided by the available resources.

Outline of This Chapter

5.2 Identifying Parallelism

- Finding parallelism is made more **difficult** because the **programming model** (which implements the software reference) is so closely associated with the **sequential compute model** of a typical processor.
- To implement parallelism from software model, we need to:
  - first move from a sequential specification to a higher level of abstraction
  - Second translate to a more concrete parallel implementation
- Nothing here is guaranteed to find the maximum parallelism in a software reference design.
  - To do that would require us to be able to infer a programmer's intent—a seemingly impossible to solve problem.



5.2.1 Ordering

- Assume a single task T composed of a sequence of n subtasks <t_0, t_1, . . . , t_{n-1}>.
- We define the following terms.
- **Totally ordered** a composed sequence the subtasks in which t_i is started before t_{i+1} for all i.
- **Legal order** of the subtasks to be an ordering that produces the same results as total order T for all inputs.
- **Partial order** a binary relation that is reflexive, antisymmetric, and transitive; useful in this chapter to describe alternative legal orderings less strict than total orderings
- The goal of finding parallelism is to take a software reference design, and find a **legal partial order** from the expressed **total order.**

Ordering: Example

- For example,

```
a = (x+y) ; // (s1)
b = (x-y) ; // (s2)
r = a/b ; // (s3)
```

- The following T and T’ are both legal orderings.
  - T = <s1, s2, s3>
  - T ‘ = <s2, s1, s3>
- Partial order:
  - “s1 has to complete before s3” and
  - “s2 has to complete before s3”
  - No specific order is imposed for s1 and s2

5.2.2 Dependence

- Defining total order mathematically:
  - Define a relation R on the set T. That is, R ⊆ T×T.
  - (t_i, t_j) ∈ R if subtask t_i **comes before** subtask t_j
    - then the total order would be **$(t_i, t_j) \in R_{tot}$ if $i < j$** or **$(\forall 0 \leq i < n \mid (t_i, t_{i+1}))$**
- What we really want is:
  - a precedence relation that is not defined by the sequence of operations in the sequential software reference design.
  - (ti, tj) ∈ R if subtask t_i **must complete** before subtask t_j
  - "**must complete**" means:
    - task ti produces some result that must be communicated to task t_j before t_j completes
    - These results are communicated through state, which is essentially the register file and primary memory (RAM) of a processor.
  - So, precedence relation: **$(t_i, t_j) \in R$ if subtask t_i writes a value that subtask t_j reads**
- Hence, The (true) dependence relation (i.e., t_j depends on t_i) is:

  $(t_i, t_j) \in D$ if **$OUT(t_i) \cap IN(t_j) \neq \emptyset$** **where**
  - OUT (t_i) the values that t_i writes
  - IN (t_i) the values that t_i reads

Dependency Example (1)

- The example experience:
  - **(True) dependence** : s7 IN set includes **r**, which is in the s5 OUT set.
  - **Output dependence:** s4 is output dependent on s6, as s5 has to complete before **a** can be overwritten.
  - **Antidependence:** the value stored in **a** has to be used by s5 before "overwritten" by s6.
    - **Can be resolved by renaming variables**
- We focus on true dependency.

```
a = (x+y) ;   // (s4)
r = a ;       // (s5)
a = (x-y) ;   // (s6)
r = r/b ;     // (s7)
```

Recall Compute Architecture:

Anti-dependency (write-after-read (WAR)) occurs when an instruction requires a value that is later updated.

Output dependency(write-after-write (WAW)) occurs when the ordering of instructions will affect the final output value of a variable.

Dependency Example (2): Generate the Dependency Set and Graph

- Consider example

1. Generate IN/OUT table:

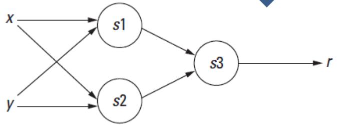
subtask t	$IN(t)$	$OUT(t)$
$s1$ `a = (x+y);`	$\{x, y\}$	$\{a\}$
$s2$ `b = (x-y);`	$\{x, y\}$	$\{b\}$
$s3$ `r = a/b;`	$\{a, b\}$	$\{r\}$

2. Compute OUT (ti)∩ IN (tj) for all i and j (except when i = j, as it does not make sense that a subtask would be dependent on itself)

	s1 OUT	s2 OUT	s3 OUT
s1 IN		$OUT(s2) \cap IN(s1) = \{\}$	$OUT(s2) \cap IN(s1) = \{\}$
s2 IN	$OUT(s1) \cap IN(s2) = \{\}$		$OUT(t_3) \cap IN(t_2) = \{\}$
s3 IN	$OUT(s1) \cap IN(s3) = \{a\}$	$OUT(s2) \cap IN(s3) = \{b\}$	

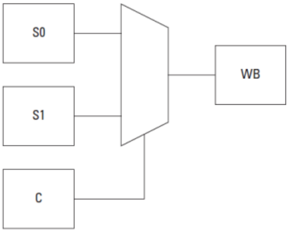
3. Dependence relation is all of the nonempty sets: D = {(s1, s3), (s2, s3)}

Graphically, the dependency is shown in this figure

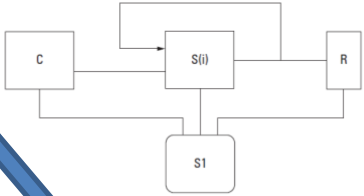


Parallelism in if/else and while-loop

```
if  ( C )  then
   S0
else
   S1
```



```
while ( C(i) ) {
    S(i)
}
```



```
while ( C(i+0) or C(i+1) or C(i+2) ) {
    if ( C(i+0) )
       S(i+0)
       if ( C(i+1) )
          S(i+1)
          if ( C(i+2) )
             S(i+2)
    i = i+2
}
```

- Parallelism in HDL constructs:
  - **If/else**: use the hardware notion of a multiplexer with the condition determining which values to propagate.
  - "**while loop**": presents a greater chance for parallelism due to loop unrolling.
  - Recognizing "induction variables" relationship to the loop control variables often lifts data dependence restrictions and permits further optimization

```
for ( i=0; i<n; i++ ) {
     a[j] = b[k];
     j++;
     k+=2;
}
```

```
_Kii2 = k;
_Kii1 = j;
for ( i = 0; i<n; i++ ) {
      a[_Kii1+i] = b[_Kii2+i*2];
}
```

5.2.3 Uniform Dependence Vectors

- **Dependence in loops** can be:
  - Within iteration
  - Between iterations.
- So, the **loop dependencies** are:
  - Loop-independent dependence
    - Dependence is within the iteration
    - But iterations are independent
  - Loop-carried dependence
    - Dependence between the iterations

Uniform Dependence Vectors: loop-independent dependence

- Dependence vectors
  - treats the body of the loop(s) as a point in space.
  - the bounds of the loop(s) define a boundary in space.
- Singly nested loop has points on a line

```
for( i=0 ; i<10 ; i++ ) {
  a[i] = c * x[i] ;
}
```

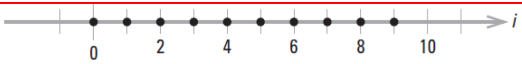


Figure 5.13. Singly nested loop iteration space.

- Doubly nested loop:
  - there are three rows and five columns.
  - Inner ***i*** loop bounds are not constants , the iteration space is not rectangular.
  - Also notice that inside the body that s2 depends on statement s1. This dependence is repeated in every iteration but it doesn't impact the execution of independent iterations.

```
for( j=1 ; j<=3 ; j++ ) {
  for( i=j ; i<=(j+5) ; i++ ) {
    c = x[j][j] ;                    // (s1)

    a[i][j] = x[i][j]/c ;            // (s2)
  }
}
```

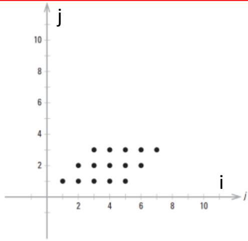


Figure 5.14. Doubly nested loop iteration space.

Uniform Dependence Vectors: loop-carried dependence

- Loop-Carried independence: single nested
  - In the below example: statement s3 depends on itself — but in a different iteration.
  - We highlight this relation between iterations with arrows
  - iteration i+1 depending on iteration i. We illustrate this as i →(i+1) for all i within the bounds of space.

```
sum = 0 ;
for( i=0 ; i<10 ; i++ ) {
   sum = sum + x[i] ; // (s3)
}
```

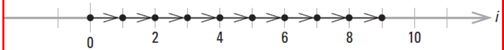
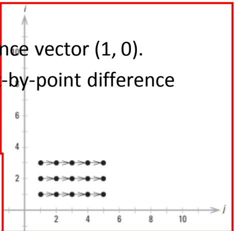


Figure 5.15. Loop-carried dependence.

- Loop-carried independence: doubly nested
  - The below example results in an iteration space with the dependence vectors
  - Inn the rows: there is the opportunity for pipeline parallelism.
  - dependence vectors in this iteration space can be summarized with the distance vector (1, 0).
  - This is because iteration (j+1, i) depends on iteration (j, i)—taking the point-by-point difference leaves (1, 0).

```
for i=1 to 3
  for j=1 to 5
    x[i] = x[i] + samp[i][j]
```



5.3. Spatial Parallelism with Platform FPGA

- Platform FPGAs
  - offer a wide variety of configurable logic resources that can be used in **parallel.**
  - offer the ability to reduce the number of integrated circuits needed in a design by migrating the logic onto the FPGA.
- This section looks
  - Looks more closely at **how to exploit the inherent parallelism** within the FPGA.
  - highlight some of the issues commonly associated with parallel implementations

Processor vs. FPGA

- The differences between a microprocessor (single-core sequential processor) and a Platform FPGA with respect to parallel processing:

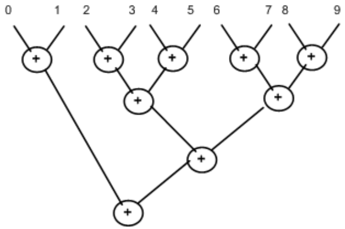
1. For tough design constraints, a single controller may not suffice.
2. An FPGA
   - provides the capability to **interface with a multitude** of **external devices**
   - Perform **intermediate and parallel** computations.

5.3.1 Parallelism Within FPGA Hardware Cores

- Expressing parallelism in HDL:
    - If/else statement
    - for-loop statement: instantiating multiple cores to operate in parallel
        - How to parallelize a loop with loop-carried dependency?

```
sum = 0 ;
for( i=0 ; i<10 ; i++ ) {
   sum = sum + x[i] ; // (s3)
}
```

- Answer: parallelize operations in hierarchy fashion.
- Speedup= 10/4 = 2.5



Control of Sequential Operations

- **Handshaking:**
  - Go/Done
  - Consumer/Producer:
    - A producer is generating data and a consumer is operating on newly generated data .
    - The consumer must indicate when it is **ready** for data, and the producer indicates when new data are **valid**.
- **FSM**: Statemachine consists of (at least) the following states:
  - **idle**: the FSM waits for some condition (perhaps the go signal) before proceeding to the running state where the operation is executed.
  - **running**: the running state can be expanded to include more than just one state; however, the state machine is never in more than one state at a time.
  - **done**: Upon completion, the FSM may signal any number of dependent components via a done signal.
- **Pipelining** : by turning each state in the state machine into individual process blocks (or separate components).
  - Implementing pipelined systems may be easier than sequential finite state machines because a designer may chain together components to build complex operations. This chain can be pipelined with the addition of registers between each component.

5.3.2 Parallelism within FPGA Designs

- How to construct systems to achieve parallelism within FPGA designs
- **Add multiple instances** of the same hardware core.
  - This is explicit parallelism; there are physically more components running in parallel.
  - However, in order to take advantage of this new compute power, we must divide the data set among the compute cores.
- At a finer granularity we can replicate **multiple instances** of a component **within a single hardware** core

Chapter-In-Review

- started with basic understanding of parallel and serial
- described a number concept related to parallelism (DOP, granularity, etc.)
- described parallelism in hardware and software
- introduced some for formal mathematics to help identify parallelism in a software reference design

Chapter 5 Terms

sequential	the same components are re-used over time and the the results are the accumulation of a sequence of operations
parallel	multiple instances of components are used and some operations may happen concurrently
totally ordered	a composed sequence of n subtasks $\langle t_0, t_1, \ldots, t_{n-1} \rangle$ in which t_i is started before t_{i+1} for all i
legal order	of subtasks to be an ordering that produces the same results as total order T for all inputs.
partial order	a binary relation that is reflexive, antisymmetric, and transitive; useful in this chapter to describe alternative legal orderings less strict than total orderings

Chapter 5 Terms

spatial in hardware designs multiple threads of control are able to operate simultaneously

fine-grain parallelism means that the subtasks t_i are relatively small, maybe the equivalent of a few instructions

coarse-grain parallelism refers to relatively large subtasks, such as an entire subroutine

degree of parallelism (DOP) can refer to: the number of concurrent operations at a single moment of time; a time-varying function over the entire application; or the average of a time-varying function over some task T which may not include the entire application