Embedded Systems Design with Platform FPGAs

Ron Sass and Andrew G. Schmidt
http://www.rcs.uncc.edu/~rsass
University of North Carolina at Charlotte
Spring 2011
Chapter 5 — Spatial Design
Chapter 5 Learning Objectives

Topics

- principles of parallelism
- using principles to find parallelism
- practical aspects of implementing on an FPGA
in electronics, parallel and serial refer to the physical arrangement of components in space i.e., two resistors can be in parallel (a) or in series (b) however, applications are typically constructed from sequences commands

with introduction of multiple processors, hardware is “in parallel” even though each processor is still programmed sequentially
Parallel and Sequential

This led to the two terms we use here:

**sequential** the same components are re-used 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

- most applications written in the 20th Century are sequential
- most embedded systems have parallel components (as well as parallel programs)
- the goal of this chapter is find parallelism in applications written sequentially
Motivating Example

- consider an application consisting of eight tasks
- each task is completed in one time-step
- with one unit, the application completes in eight cycles
Before and After Finding Parallelism

(a) u0: unit

(b) u0: unit

(c) u1: unit

(d) u2: unit

(e) u3: unit

\[ t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \]
If we had four units, we might be able to complete the application in few cycles.

- some units may get two tasks
- some units may get one task
- another unit may get three task

This results in the application finishing $2.1 \times$ faster but requires $4 \times$ as much resources.
One might ask: Why not move task $t_7$ to unit $u3$?
- the work will be evenly distributed, and
- the application will complete in 2 versus 3 cycles...
- however, the tasks may have ordering requirements
- $t_6$ has to complete before $t_7$ begins
- and $t_2$ has to complete before $t_6$ begins

Finding the legal parallelism in an application is our goal.
Over the next few slides, we’ll discuss some key parallel concepts.

- granularity
- degree of parallelism
- spatial organizations
Spatial Design

Granularity

The size of the parallel tasks influences many decisions.

The granularity is typically determined by the frequency at which the tasks communicate: fine-grain tasks generally need to communicate more frequently.
Degree of Parallelism

degree of parallelism (DOP) can be used in multiple senses

or,

The DOP in application is useful when considering resource utilization versus maximum performance.
With the DOP illustrated, we can reason about potential solutions.

For example,

- since the maximum $DOP(t)$ is 6, having more than six units would be overkill

- using four units would achieve 78% of the maximum performance while only using 66% of the resources
Flynn’s Taxonomy

One well-known way of classifying parallel architectures is by how many streams of instructions and how many streams of data are possible.

- Single Instruction Stream, Single Data Stream (SISD)
- Single Instruction Stream, Multiple Data Stream (SIMD)
- Multiple Instruction Stream, Single Data Stream (MISD)
- Multiple Instruction Stream, Multiple Data Stream (MIMD)
Single Instruction Stream, Single Data Stream (SISD) is the single-core, von Neumann style processor model used in many general-purpose computers:

- one stream of instructions
- one stream of data read/write’s
Single Instruction Stream, Multiple Data Stream (SIMD) model uses simple Processing Elements (PEs) that are all performing instructions from a single instruction stream in lock step:

- one stream of instructions broadcasted to all PEs
- each PE had its own memory (so multiple streams of data being simultaneously read and written)
Multiple Instruction Stream, Single Data Stream (MISD) here parallel units each have their own, independent instruction stream but a single stream of data travels from unit to unit

- usually data travels over an explicit communication
- sometimes called “systolic architecture”
Multiple Instruction Stream, Multiple Data Streams (MIMD)
(probably the most common) multiple processing units, each with its own:

- memory hierarchy,
- stream of instructions, and
- stream of data to/from local memory
At one point in the late 1990s, commodity off-the-shelf components made MIMD very economical and thus very popular.

- volume of parallel processors was MIMD hardware but programmed as Single Program Mutiple Data (SPMD)
- that is, one program was replicated to work on multiple data streams
- however, replicated copies did not run lock step
- also, internally GPGPUs and certain graphics instructions on modern processors are similar to SIMD

Also, FPGAs are large enough now it is feasible to implement any of these architectures.
thus far, we have considered parallel hardware structures to utilize multiple instruction streams or multiple data streams, we must find them in our software reference design

- vector and pipeline parallelism
- control parallelism
- data (regular) parallelism
suppose we have a task $T$ that can be divided up into $n$ subtasks $\{t_0, t_1, ..., t_{n-1}\}$

(the granularity can vary)

if we map these subtasks onto parallel units as illustrated below

![Diagram](image)

each task performs part of the processing and forwards its partial results to the next subtask via a buffer
clearly, if we only need to produce one output, the performance is \( n \) cycles per \( T \) operation

however, if we need a sequence of \( m \) outputs

\[
O = \langle o_0, o_1, ..., o_{m-1} \rangle
\]

then \( m \) number of \( T \) operations is

\[
\frac{m}{m + n}
\]

and clearly as \( m \to \infty \), the average performance approaches 1 cycle per \( T \)
one may not think of “large vectors of data” in embedded systems

however, many embedded systems *do* periodically sample their environment

this results in a large stream of inputs (even though the data was never stored as a vector in primary or secondary memory)
in the simplest situation, all of the pipeline’s parameters are “in balance”
- every task takes the same amount of time
- every partial result requires same amount memory to communicate it
- arrival rate of input matches rate at which the pipeline can process it

if not,
- time-space diagram may be helpful in identifying the bottleneck (see next slide)
- one may need to analyze the bandwidth (see next chapter)
Time-Space Diagram

Thread 3
- Active
  - C A

Thread 2
- Idle
  - A
- Comm

Thread 1
- Comm
  - A I A

Thread 0
- Active
  - C A
- Idle

Time
Control Parallelism

control parallelism (also known as irregular or \textit{ad hoc} parallelism)

- examines a single stream of instructions looking to separate them into parallel operations
- in some ways this can be considered a a generalization of pipeline parallelism
  - pipeline parallelism tends to be more uniform
  - control parallelism typically has a much more complex communication patterns
- the number of cycles to complete each parallel tasks in control parallelism tends to vary more than pipeline parallelism
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 code would be:

(s1) \hspace{1cm} descr = sqrt(b*b-4*a*c)

(s2) \hspace{1cm} denom = 2*a

(s3) \hspace{1cm} x0 = (-b+descr)/denom

(s4) \hspace{1cm} x1 = (-b-descr)/denom

The inputs to our entity will be $a$, $b$, and $c$. The outputs will be $x0$ and $x1$. 
Statement-Level Granularity

\[ x_0 = \frac{2a}{\sqrt{b^2 - 4ac}} \]

\[ x_1 = \frac{-b + \text{descr}}{\text{denom}} \]

\[ x_0 = \frac{2a}{\sqrt{b^2 - 4ac}} \]

\[ x_1 = \frac{-b + \text{descr}}{\text{denom}} \]
Operational-Level Granularity
Data (Regular) Parallelism

- in contrast to control parallelism, data parallelism looks at a single data stream and tries to separate it into multiple, parallel data streams
- advantage: dividing up large datasets can often lead to a higher degree of parallelism (and thus higher performance)
- disadvantage: often the entire dataset is required all at once
Example of Data Parallelism

- Starting with some array of data,

  
  

  we want to assign each parallel unit a portion of the data

- it can be divided up in a blocked arrangement

  
  

  or it can be divided up in a cyclic fashion

  
  

Identifying Parallelism

Main challenges:

- software reference design most likely written with a sequential model in mind
- determining granularity of parallelism
- finding legal, alternative orderings so that some operations can execute concurrently

Usually this requires thinking about the algorithm more abstractly before implementing in hardware.
Orderings

- consider a task $T$ composed of a sequence of $n$ subtasks:
  \[ T = \langle t_0, t_1, \ldots, t_{m-1} \rangle \]

- we say that $T$ is **totally ordered** because

- programs written for a single processor are written as a total order

- we define a **legal order** as
Example of Orderings

- suppose task $T$ is the following code
  
  ```
  a = (x+y) ; // (s1)
  b = (x-y) ; // (s2)
  r = a/b ; // (s3)
  ```

- the original total ordering is
  
  $$T = \langle s1, s2, s3 \rangle$$

- another legal ordering is
  
  $$T' = \langle s2, s1, s3 \rangle$$

- this demonstrates the main concept
  - “$s1$ has to complete before $s3$” and
  - “$s2$ has to complete before $s3$”

  but $s1$ and $s2$ are independent

  independent operations can be executed in parallel
we want to take a totally ordered, sequential software reference design and find a partial ordering of its subtasks

formally, a partial order is a set (our subtasks) and a binary relation ("must complete before")\(^1\)

in our case, this relation has a specific name called dependence

\(^1\)technically, this is precedence relation (but they are very similar and the partial order is more common)
given a sequence $T$ we can define a relation $R$ on the set $T$

$$R \subseteq T \times T$$

in English, the relation is

$$(t_i, t_j) \in R \text{ if subtask } t_i \text{ comes before subtask } t_j$$

and we could recreate the total order of a sequential program with

$$(t_i, t_j) \in R_{\text{tot}} \text{ if } i < j$$

or just

$$(\forall 0 \leq i < n | (t_i, t_{i+1}))$$

however, to find other legal orderings what we want to do is find a subset of $R_{\text{tot}}$

we determine if it is legal to remove the ordering by looking at the IN and OUT sets of the subtasks
these concepts can be applied at any granularity but to illustrate, we’ll use program statements here

the OUT set of a task $t_i$, denoted $OUT(t_i)$, is all of the values that $t_i$ outputs

the IN set of a task $t_i$, denoted $IN(t_i)$, is all of the values that $t_i$ requires

for statements,

- the OUT set is (approximately) the memory location of the left-hand side of an assignment statement
- the IN set are the memory locations of all of the variables reference on the right-hand side
with the IN sets and OUT sets of every subtask, we can formally define the dependence relation as

$$(t_i, t_j) \in D \quad \text{if} \quad OUT(t_i) \cap IN(t_j) \neq \emptyset$$

note: we talked about values when introducing the IN sets and OUT sets

- variables may get rewritten over time — we want to track the values not just variable names
- references and aliases can confound the problem because we cannot always tell what address a pointer has
- array references with variable indices are often solvable but can be tricky for humans
by tracking values, we finding what is called true dependence

consider the following code

\[
\begin{align*}
\text{a} &= (x+y) \; ; \quad \text{(s4)} \\
\text{r} &= \text{a} \; ; \quad \text{(s5)} \\
\text{b} &= (x-y) \; ; \quad \text{(s6)} \\
\text{r} &= \text{r}/\text{b} \; ; \quad \text{(s7)}
\end{align*}
\]

some places will say there is output dependence between s5 and s7 (because s7 has to execute after s5 to produce the same results)

we also have to be sure s5 is executed before s6 because

these other forms of dependence can usually be avoided by simply renaming the variables or converting to the code to another form
consider the previous code re-written in three statements

\[
a = (x+y) ; \quad // \quad (s4) \\
b = (x-y) ; \quad // \quad (s6) \\
r = a/b ; \quad // \quad (s7)
\]

we can calculate the IN sets and OUT sets for all three:

<table>
<thead>
<tr>
<th>subtask  $t$</th>
<th>$IN(t)$</th>
<th>$OUT(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = (x+y)$</td>
<td>${x, y}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$b = (x-y)$</td>
<td>${x, y}$</td>
<td>${b}$</td>
</tr>
<tr>
<td>$r = a/b$</td>
<td>${a, b}$</td>
<td>${r}$</td>
</tr>
</tbody>
</table>
next, we can manually compute the dependence relation

\[(t_i, t_j) \in D \quad \text{if} \quad \text{OUT}(t_i) \cap \text{IN}(t_j) \neq \emptyset\]

via brute-force:

<table>
<thead>
<tr>
<th></th>
<th>OUT (s1) (\cap) IN (s1)</th>
<th>OUT (s2) (\cap) IN (s3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>OUT (s2) (\cap) IN (s1) = {}</td>
<td>OUT (s2) (\cap) IN (s1) = {}</td>
</tr>
<tr>
<td>s2</td>
<td>OUT (s1) (\cap) IN (s2) = {}</td>
<td>OUT (t_3) (\cap) IN (t_2) = {}</td>
</tr>
<tr>
<td>IN</td>
<td>OUT (s1) (\cap) IN (s3) = {a}</td>
<td>OUT (s2) (\cap) IN (s3) = {b}</td>
</tr>
</tbody>
</table>
All of the non-empty intersections indicate that there is a dependence from one subtask to another.

In the previous example, this would be:

\[ D = \{(s1, s3), (s2, s3)\} \]

If you treat \(x\) and \(y\) as “source” components and \(r\) as a sink, these dependences can be visualized (above).
So far, we have only considered sequences of assignment statements; imperative programming languages also have conditional and iterative control structures.

For example, in the pseudocode

\[\text{if}( \ i < j \ ) \ \text{then} \]
\[x = y;\]

whether \( x \) depends on \( y \)'s value is indirectly influenced by the expression \( i < j \)

When translating to hardware, we can use templates to handle this form of dependence without formalizing it.
rather than exhaustively go through every control structure that might be found in a software reference design, we focus on two general cases (the rest can be derived from these)

- if-then-else
- while loop

for both, we illustrate with a single assignment statement but this could be a sequence of statements or other nested control statements
If-Then-Else

- an if-then-else structure can be created by instatiating
  - “then” statement
  - “else” statement
  - and the “conditional” expression

- the OUT set values of the “then” and “else” parts are the inputs to a MUX and the conditional determines which values propagate forward
If (C) then
  S0
else
  S1

if  ( C )  then 
 S0 
else 
 S1
a while structure can be created by instatiating
  “body”
  and the “conditional” expression
next, a state machine is needed to determine when the conditional evaluates to false
during an iteration, some values will write back to an external store (not shown) or feed back for the next iteration
while ( C(i) ) {
    S(i)
}

While-Loop (Graphically)
Loops In General

- loops are a rich source for parallelism in hardware
- very large body of research into how to formally analyze loops
- a full discussion is beyond the scope of this chapter but a few techniques are highlighted
Loop Unrolling

- loop-carried dependences are dependences between the iterations of the loop
- if there are no loop-carried dependences, unrolling is simply mechanical

```plaintext
while ( C(i) ) {
  S(i)
  i = i+1
}
```

```plaintext
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
}
```

- sometimes the conditionals can be avoided
Loop Induction Variables

- often programmers will introduce loop-carried dependences to reduce the amount of computation inside the loop
- often in embedded systems, the FPGA provides plenty of computation and we would prefer to avoid the dependence
- below shows an example of removing induction variables

Before:
```c
for ( i=0; i<n; i++ )
    a[j] = b[k];
    j++;
    k+=2;
```

After:
```c
_Kii2 = k;
_Kii1 = j;
for ( i = 0; i<n; i++ )
    a[_Kii1+i] = b[_Kii2+i*2];
```
as the number of iterations in a loop increase, the size of the set $D$ increases

if the relations between the iterations are uniform, then we can use a more compact form

uniform dependence vectors also help the embedded system designer see the relationship between the iterations graphically

for a singly-nested loop, the iterations are laid out on a line, doubly-nested loops on an $x – y$ plane, and so on
Spatial Design

Iteration Spaces

Singly-Nested Loop

Doubly-Nested Loop
sum = 0;
for( i=0 ; i<10 ; i++ )
    sum = sum + x[i]; // (s3)
for \( i = 1 \) to 3

for \( j = 1 \) to 5

\[
x[i] = x[i] + samp[i][j]
\]
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 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 $⟨t_0, t_1, ..., t_n-1⟩$ 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
Spatial Design

Chapter 5 Terms

degree of parallelism (DOP)