18-643 Lecture 9: C-to-HW Synthesis

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: develop a mental model for what C-to-HW must do on your behalf

• Notices
  – Handout #4: lab 2, due noon, 10/7
  – 4 weeks to project proposal!!!!

• Readings
  – Ch 7, Reconfigurable Computing
  – skim: IEEE Design & Test of Computers, No. 4, Jul 2009. Special Issue on High-Level Synthesis
A Program is a Functional-Level Spec

```c
int fibi(int n) {
    int last=1; int lastlast=0; int temp;

    if (n==0) return 0;
    if (n==1) return 1;

    for(;n>1;n--) {
        temp=last+lastlast;
        lastlast=last;
        last=temp;
    }

    return temp;
}
```
A Program is a Functional-Level Spec

```c
int fibm(int n) {
    int *array,*ptr; int i;
    if (n==0) return 0;
    if (n==1) return 1;
    array=malloc(sizeof(int)*(n+1));
    array[0]=0; array[1]=1;
    for(i=2,ptr=array ; i<=n ; i++,ptr++)
        *(ptr+2)=*(ptr+1)+*ptr;
    i=array[n];
    free(array);
    return i;
}
```
A Program is a Functional-Level Spec

```c
int fibr(int n) {
    if (n==0) return 0;
    if (n==1) return 1;

    return fibr(n-1)+fibr(n-2);
}
```
Opening Questions

• Do they all compute the same “function”?

• Should they all lead to the same hardware?

• Should they all lead to “good” hardware?
  – what does recursion look like in hardware?
  – what does `malloc` look like in hardware?
What is in a C Function?

• What it specifies?
  – abstracted data types (e.g., int, floats, doubles)
  – step-by-step procedure to compute the return value from input arguments
  – a sequentialized execution

• What it doesn’t specify?
  – encoding of the variables
  – where the state variables are stored
  – execution timing, neither in terms of wall-clock time, clock cycles, or instruction count
  – what types and how many functional units to use
  – what is strictly necessary for correctness
Mapping Program to Hardware

• For you to produce “good” structural RTL
  – control low level details
  – unrestricted design freedom
  – massive concurrency

• C-to-HW (i.e., C-to-RTL) compiler bridges the gap between functionality and implementation
  – fill in the details below the functional abstraction
  – make good decisions when filling in the details
  – extract parallelism from a sequential specification

Keep in mind: what you don’t need to specify you also can’t control
A Look at Scheduling and Allocation
Procedural Block to Data Flow Graph

static elaboration to single-assignment

static elaboration to single-assignment
Data Flow Graph

- Captures data dependence irrespective of program order
  - nodes = operator
  - edge = data flow
- "Work" is total delay if done sequentially
  - e.g., if delay(+) = 1, delay(*) = 2, work = 6
- "Critical path" is the longest path from input to output
  - e.g., critical path delay = 4
  - no implementation can complete faster than critical path delay
- Combinational or sequential??

\[
\begin{align*}
  v &= a + b; \\
  w &= b \times c; \\
  x &= v + c; \\
  y &= v + w; \\
  z &= x + y;
\end{align*}
\]
Program-Order Sequential Mapping

- Need only one of each functional unit type: 1 adder, 1 multiplier
- Delay equal “work”: 6

In contrast, if combinational
- 4 adder, 1 multiplier
- delay=4

Is there a shorter schedule for 1 adder and 1 multiplier?
Optimized Sequential Mapping

• In general,
  – given a set of functional units, what is the shortest schedule
  – given a schedule, what is the minimum set of functional units
  – given a target delay (>= critical path), find a min-cost schedule

• Very efficient algorithms exist for solving the above

• Harder part is setting the right goal
  – minimum delay could be expensive
  – minimum resource could be slow

\[
\begin{align*}
\text{delay} & = 4 \text{ using 1 adder and 1 multiplier}
\end{align*}
\]
Generating Datapath

How do I know 3 registers is enough?
Control FSM

- Assume initially a in r1; b in r2; c in r3

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>add</th>
<th>mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel1</td>
<td>en1</td>
<td>sel2</td>
<td>en2</td>
<td>sel3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>add</td>
<td>1</td>
<td>mul</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
It should remind you of this

inputs

FSM

datapath

outputs

clock
Good Hardware Needs Parallelism
Where to Find Parallelism in C?

- C-program has a sequential reading
- Scheduling exploits operation-level parallelism in a basic block ($\approx$ work/critical-path-delay)
  - “ILP” is dependent on scope
  - techniques exist to enlarge basic blocks and to increase operation-level parallelisms: loop-unrolling, loop pipelining, superblock, trace scheduling, etc.

Many ideas first developed for VLIW compilation
- Structured parallelism can be found across loop iterations, e.g., data parallel loops
Loop Pipelining

\[
\begin{align*}
&\text{for}(i=0; i<N; i++) \\
&\quad \{ \\
&\quad \quad v = a[i] + b[i]; \\
&\quad \quad w = b[i] \times c[i]; \\
&\quad \quad x = v + c[i]; \\
&\quad \quad y = v + w; \\
&\quad \quad z[i] = x + y; \\
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\text{for}(i=1; i<N; i++) \\
&\quad \{ \\
&\quad \quad v' = v; \ w' = w; \\
&\quad \quad v = a[i] + b[i]; \\
&\quad \quad w = b[i] \times c[i]; \\
&\quad \quad x = v' + c[i-1]; \\
&\quad \quad y = v' + w'; \\
&\quad \quad z[i-1] = x + y; \\
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\text{for}(i=2; i<N; i++) \\
&\quad \{ \\
&\quad \quad v' = v; \ w' = w; \\
&\quad \quad x' = x; \ y' = y; \\
&\quad \quad v = a[i] + b[i]; \\
&\quad \quad w = b[i] \times c[i]; \\
&\quad \quad x = v' + c[i-1]; \\
&\quad \quad y = v' + w'; \\
&\quad \quad z[i-2] = x' + y'; \\
&\quad \}
\end{align*}
\]
Pipelined Loop

```c
for(i=2;i<N;i++) {
    v' = v;  w' = w;
    x' = x;  y' = y;

    v = a[i]+b[i];
    w = b[i]*c[i];

    x = v'+c[i-1];
    y = v'+w';

    z[i-2]= x'+y';
}
```

- In SW, loop pipelining increases producer-consumer distance
- In HW, work on parts of 3 different iterations in same cycle
Loop Unrolling

for (i=0; i<N; i++)
{
    v = a[i]+b[i];
    w = b[i]*c[i];
    x = v+c[i];
    y = v+w;
    z[i] = x+y;
}

for (i=0; i<N; i+=2)
{
    v = a[i]+b[i];
    w = b[i]*c[i];
    x = v+c[i];
    y = v+w;
    z[i] = x+y;
    v_ = a[i+1]+b[i+1];
    w_ = b[i+1]*c[i+1];
    x_ = v+c[i+1];
    y_ = v_+w_;
    z[i+1] = x_+y_;
}

Follow by other optimizations, including pipelining
Not only about compute

\[
\begin{align*}
\text{for}(i=0; \ i<N; \ i++) \\
\text{for}(j=0; \ j<N; \ j++) \\
\text{for}(k=0; \ k<N; \ k++) \\
C[i][j] &= C[i][j] + A[i][k] \cdot B[k][j]
\end{align*}
\]

Inner-most loop’s iterations have dependency through \(C[i][j]\)
Loop Interchange

\[
\begin{align*}
&\text{for}(k=0; \ k<N; \ k++) \\
&\quad \text{for}(i=0; \ i<N; \ i++) \\
&\quad \quad \text{for}(j=0; \ j<N; \ j++) \\
&\quad \quad \quad C[i][j]=C[i][j]+A[i][k]*B[k][j]
\end{align*}
\]

Parallelizable over the 2-innermost loops
Refactor for Parallelism

for\(i=\ldots\)
for\(j=\ldots\)

\[C[i,j] += f(i,j,0) + f(i,j,1) + \ldots + f(i,j,k-1)\]

parallel kernel pipelines

fully unrolled inner loops

for\(k=\ldots\)

\[C[i,j] += f(i,j,k)\]
C-to-HW is Very Real Today

• Many commercial and research tools are available
  – most major CAD vendors
  – Xilinx Vivado and Intel OpenCL
  – ROCCC [UC Riverside] and LegUP [U Toronto] (free)
  – LLVM makes it pretty easy to roll-your-own

• State of technology
  – work very well on some domain or applications
  – not without blindspots
  – human-in-the-loop pragmas important

Useful in different ways to an expert HW designer and a so-so HW designer
Parting Thoughts

• C-to-HW compiler fills in details between algorithm and implementation
  – front-end (not covered here) can use standard optimizations (deadcode, common-subexp, strength-reduction....)
  – back-end shares many techniques with VLIW and parallelizing compilers

• No magic—good HW only if it is in the program
  – not every computation is right for HW so not every C-program is right for HW
  – even for right ones, how the C is written matters
Tortoise and Hare

- Tortoise
  - delivers exact optimal implementation to a fully specified objective (functional + tuning)
  - perfection takes time
    - say last 10% of quality takes up 90% of the time

- Hare
  - only gets to 90% quality
  - delivers the design 10 times faster

This hare doesn’t take a nap after one design . . .
The Design Race

- Good Enough Box
- out-of-time
- 90%
- 1/perf
- hey, it works
- educated guess
- best possible
Why the Hare Wins

• In real design projects
  – don’t always know exact target initially
  – can’t land first shot on target anyway
  – good enough really is good enough
  – hitting schedule is everything
show at COMDEX in Nov or bust in Dec

• There are a lot more rabbits than turtles in this world; there are not enough turtles in this world

Even more turkeys . . . but that’s a different class

All characters appearing in this story are fictitious. Any resemblance to real persons, living or dead, is purely coincidental.