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 C-to-HW synthesis

• Notices
  – Handout #4: lab 2, due noon, 10/6
  – 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

• Recall why RTL design is hard
  – reason #1: low level abstraction
  – reason #2: unrestricted design freedom
  – reason #3: 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

\[
\begin{align*}
\{ & \\
& x = b; \\
& \text{if (y)} \\
& \quad x = x + a; \\
& \}
\end{align*}
\]

\[
\begin{align*}
\{ & \\
& x_1 = b; \\
& \text{if (y)} \\
& \quad x_2 = x_1 + a; \\
& \quad \text{else} \\
& \quad \quad x_2 = x_1 \\
& \quad \}
& x = x_2
\end{align*}
\]

static elaboration to single-assignment

same from RTL synth
Data Flow Graph

- Captures data dependence irrespective of program order
  - nodes = operator
  - edge = data flow

  Corresponds to a combinational mapping

- “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 schedule can produce delay below critical path delay
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

delay=4 using 1 adder and 1 multiplier
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></th>
<th>r1</th>
<th></th>
<th>r2</th>
<th></th>
<th>r3</th>
<th></th>
<th>add</th>
<th></th>
<th>mult</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sel1</td>
<td>en1</td>
<td>sel2</td>
<td>en2</td>
<td>sel3</td>
<td>en3</td>
<td>sel5</td>
<td>sel6</td>
<td>sel7</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>r1</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>add</td>
<td>1</td>
<td>mul</td>
<td>1</td>
<td>-</td>
<td>r1</td>
<td>r3</td>
<td>r2</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>r1</td>
<td>r3</td>
<td>-</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>r2</td>
<td>r1</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, superblocl, trace scheduling, etc.

  Many ideas first developed for VLIW compilation

• Structured parallelism can be found across loop iterations, e.g., data parallel loops
Aside: “ILP” is dependent on scope

- Average ILP = \( \approx \) work/critical-path-delay

\[
\begin{align*}
    r1 & \leftarrow r2 + 1 \\
    r3 & \leftarrow r1 / 17 \\
    r4 & \leftarrow r0 - r3 \\
    r11 & \leftarrow r12 + 1 \\
    r13 & \leftarrow r19 / 17 \\
    r14 & \leftarrow r0 - r20
\end{align*}
\]

ILP=1

ILP=2
Loop Pipelining

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

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

\[
\begin{align*}
\text{for}(i=2; i < N; i++) & \{ \\
v' &= v; \\
w' &= w; \\
x' &= x; \\
y' &= y; \\
v &= a[i] + b[i]; \\
w &= b[i] * c[i]; \\
x &= v' + c[i-1]; \\
y &= v' + w'; \\
z[i-2] &= x' + y'; \\
z[i-1] &= x + y; \\
\} \\
\end{align*}
\]
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_;
Not only about compute

```c
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

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

for (k=0; k<N; k++)
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j]

Parallelizable over the 2-innermost loops
Example: Perfect Loop Nests

• Matrix-matrix multiply

\[
\text{for}\ (i=\ldots
\quad \text{for}\ (j=\ldots
\quad \quad \text{for}\ (k=\ldots
\quad \quad \quad C[i][j]+=A[i][k]*B[k][j]
\]

• Goal
  – expose parallelism
  – data reuse to avoid DRAM access
  – improve DRAM access pattern
Loop Nests to Streams

for (i=...)
  for (j=...)
    for (k=...)
      C[i][j] += A[i][k]*B[k][j]

turn indexed memory references to streams

ROCCC: loop nest to pipelined streaming kernel
[www.jacquardcomputing.com]
Control for Each Stream

for (i=... for (j=... for (k=... GET A[i][k]

for (i=... for (j=... for (k=... GET B[k][j]

for (i=... for (j=... for (k=... GET C[i][j]
READ SYNC

for (i=... for (j=... for (k=... PUT C[i][j]
WRITE SYNC

Synchronization on C[i][j] due to RAW dependence
Refactor for Parallelism

parallel kernel pipelines

fully unrolled inner loops
Recognize Accumulators

\[
\begin{align*}
\text{for}(k=\ldots & \quad \text{for}(k=\ldots \\
\text{for}(i=\ldots & \quad \text{for}(i=\ldots \\
\text{for}(j=\ldots & \quad \text{for}(j=\ldots \\
\text{GET } C[i][j] & \quad \text{PUT } C[i][j] \\
\text{READ SYNC} & \quad \text{WRITE SYNC}
\end{align*}
\]
Cache Repeated Data

```plaintext
for(k=...
for(i=...
for(j=...
GET A[i][k]
```

repeat
Coalesce Sequential Transfers

```
for(k= ... 
    for(i=... 
        for(j=... 
            GET B[k][j]

    for(k=... 
        for(i=... 
            GET-ROW B[k]

```
Strided Accesses

for (k=... 
  for (i=... 
    [for (j=... ]
    GET A[i][k]

strided accesses
bad for caches and
DRAM row-buffer
Fetch Blocks of Rows

\[
\text{for (k=... } \\
\text{ for (i=... } \\
\text{ [for (j=... ]} \\
\text{ GET A[i][k]}
\]

Work on blocked regions
- access DRAM in rows
- buffer multiple rows
- permute data on-the-fly
to form strided column stream
C-to-HW is Very Real Today

- Many commercial and research tools are available
  - most major CAD vendors
  - Xilinx Vivado and Altera 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