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
  – Project status report due each Friday
  – Handout #6: lab 2, due noon, Monday, 10/12

• 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

```
{ 
  x = b;
  if (y) 
    x = x + a;
}

{ 
  x₁ = b;
  if (y)
    x₂ = x₁ + a;
  else
    x₂ = x₁
  x = x₂
}
```

static elaboration to single-assignment

same from RTL synth
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??

```
v = a + b;
w = b * c;
x = v + c;
y = v + w;
z = x + y;
```
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

\[ \text{delay} = 4 \text{ using 1 adder and 1 multiplier} \]
How do I know 3 registers are needed?
Control FSM

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

<table>
<thead>
<tr>
<th></th>
<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>
<td>en3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>add</td>
<td>1</td>
<td>mul</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
It should remind you of this

inputs

FSM

datapath

clock

outputs
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 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_;
}

work=?? critical path=??
Loop Pipelining

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=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;
}

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';
}

v = a[0]+b[0];
w = b[0]*c[0];
x = v+c[0];
y = v+w;

v = a[1]+b[1];
w = b[1]*c[1];

Pipelined Loop

In SW, loop pipelining increases producer-consumer distance.

In HW, work on parts of 3 different iterations in same cycle.

```
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';
}
```

`work=?? critical path=??`
Pipelined Loop

```
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
A Look at dependency & memory access

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

*assume row-major and large 2-power N*
Loop Reordering

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

Data-parallel over the i and j loops

not associative for float
How to structure memory and array layout?

\[ C[i][j] += f(i, j, k) \]

\[ \text{pipelined kernel} \]

\[ \begin{align*}
  &a[i][k] \\
  &b[k][j] \\
  &c[i][j] \quad + \\
  &c[i][j]
\end{align*} \]

\[ \text{unrolled inner loops} \]

\[ \begin{align*}
  &a[i][0] \\
  &b[0][j] \\
  &a[i][1] \\
  &b[1][j] \\
  &a[i][k-1] \\
  &b[k-1][j] \\
  &c[i][j] \quad + \\
  &c[i][j] \\
  &c[i][j] \\
  &c[i][j] + f(i, j, 0) + f(i, j, 1) \\
  &\ldots + f(i, j, k-1)
\end{align*} \]
Memory not Monolithic Abstraction

- Control memory organization to match access pattern

```
<table>
<thead>
<tr>
<th>wdata_0</th>
<th>addr_0</th>
<th>we_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>wdata_1</td>
<td>addr_1</td>
<td>we_1</td>
</tr>
<tr>
<td>wdata_*</td>
<td>addr_*</td>
<td>we_*</td>
</tr>
</tbody>
</table>
```

```
RRAM

wdata
addr
we
Rdata
```

width
height
Control over Data Layout

- An array of $N$ words; index is $\lg_2 N$ bits
- $N$-word total storage
  - divided into $B$ banks; bank number is $\lg_2 B$ bits
  - each bank is $W$-word wide; word-offset is $\lg_2 W$ bits
  - index within bank is $\lg_2 (N/B/W)$ bits
- Objectives
  - maximize spatial locality in green
  - maximize entropy in red

Compilers can infer some of this but not all cases

In general interleaved & reordered
Example: Image Frame

- $N$ pixels in $\sqrt{N}$-by-$\sqrt{N}$ frame
- Spatial locality in $\sqrt{W}$-by-$\sqrt{W}$ tiles
- Parallelism across same-column tiles

$\lg_2 N$ pixel index

- Use for $\lg_2 B$
- Use for $\lg_2 N/B/W$
- Use for $\lg_2 W$

Can you tell the compiler (through C) this is what you want?
Parting Thoughts

- C-to-HW compiler fills in details between algorithm and implementation
- 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
- C-to-HW technology is very real today
  - work very well on some domain or applications
  - has blindspots; need human-in-the-loop pragmas

*Useful in different ways to an expert HW designer vs. a so-so HW designer vs. a SW programmer*