18-643 Lecture 7: C-to-HW Synthesis:

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: develop a mental model for how to turn “proper” C into “proper” HW, whether by a compiler or by hand

• Notices
  – Handout #4: lab 1, due noon, 9/25
  – Project status report due each Friday

• Readings (see lecture schedule online)
  – for textbook treatment: Ch 7, Reconfigurable Computing
C as Model of Computation for HW?

- Common arguments for using C to design HW
  - easy algorithm specification
  - popularity, popularity, popularity
- A large semantic gap to bridge
  - sequential thread of control
  - abstract time
  - abstract I/O model
  - missing structural notions: bit width, ports, modules
  - reactive execution
- No problem getting HW from C, but good HW?

All sequential, imperative languages
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);
}
```
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 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;
}
```
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)
  – operators and step-by-step procedure to compute the return value from input arguments
  – a sequential execution

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

• For you to produce “good” structural RTL
  – identify suitable “temporal and spatial pattern”
  – flesh out concrete datapath (bit/cycle exact)
  – develop correct and efficient control sequencing

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

*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;
}

Static elaboration to single-assignment

{  
  x1 = b;
  if (y)
    x2 = x1 + a;
  else
    x2 = x1
  x = x2
}
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??
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 are needed?
Control FSM

- Assume initially \( a \) in \( r1 \); \( b \) in \( r2 \); \( c \) in \( r3 \)

<table>
<thead>
<tr>
<th>( \text{r1} )</th>
<th>( \text{r2} )</th>
<th>( \text{r3} )</th>
<th>( \text{add} )</th>
<th>( \text{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 Concurrency
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;
}

data-parallel iterations

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];
x = v+c[1];
y = v+w;

v = a[2]+b[2];
w = b[2]*c[2];
x = v+c[2];
y = v+w;
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

work=?? critical path=??
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

This looks more familiar?
How Hard is MMM?

float A[N][N], B[N][N], C[N][N];

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

- $T_1$, $T_\infty$, $P_{avg} = T_1 / T_\infty$?
- # of memory access?
- $T_1$ / # of memory access?
What is all not said in the code?
A Look at dependency & memory access

```
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]
```
Loop Reordering

\[
\text{for}(k=0; \ k<N; \ k++) \\
\text{for}(i=0; \ i<N; \ i++) \\
\text{for}(j=0; \ j<N; \ j++) \\
C[i][j] \ += \ A[i][k]\times B[k][j]
\]

*Data-parallel over the i and j loops*
Loop Reordering Affects Parallelism

$O(N^3)$ memory access necessary?

What about strided $b$ access?

$C[i][j] += f(i, j, k)$
Memory not Monolithic Abstraction

• Control memory organization to match access pattern

```
bank 0
  wdata_0
  addr_0
  we_0
  rdata_0

bank 1
  wdata_1
  addr_1
  we_1
  rdata_1

bank *
  wdata_*
  addr_*
  we_*
  rdata_*
```
Control over Data Layout

• An array of N words; index is $\log_2 N$ bits
  
  $\log_2 N$ array index (sequential)

• N-word total storage
  – divided into $B$ banks; bank number is $\log_2 B$ bits
  – each bank is $W$-word wide; word-select is $\log_2 W$ bits
  – line index within bank is $\log_2 (N/B/W)$ bits

• Assign bank #, word select and index to maximize
  – spatial locality
    in word select
  – “entropy” in bank #

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 different-row tiles

Can you tell the compiler (through C) this is what you want?
A Small Concrete Example: N=16, W=4

- **Pixel idx**: $a_3a_2a_1a_0$
- **Column idx**: $a_3a_2a_1a_0$
- **Row idx**: $a_3a_2a_1a_0$
- **Tile idx**: $a_3a_2a_1a_0$
- **Word sel**: $a_3a_2a_1a_0$
- **Bank offset**: $a_3a_2a_1a_0$
- **Bank #**: $a_3a_2a_1a_0$

**Diagram:***

- **Width=4 pixels**
- **Height=2**

- **Bank 0**
  - `p0000`, `p0010`, `p0100`, `p0110`, `p1000`, `p1010`, `p1100`, `p1110`

- **Bank 1**
  - `p0001`, `p0011`, `p0101`, `p0111`, `p1001`, `p1011`, `p1101`, `p1111`

**Notes:**
- **Whole frame at a time**
- **Concurrent access to different row tiles**

---

18-643-F23-L07-S32, James C. Hoe, CMU/ECE/CALCM, ©2023
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**