BeiHang Short Course, Part 3: C-to-Hardware Synthesis
(Disclaimer: this topic is not my research area)

James C. Hoe
Department of ECE
Carnegie Mellon University

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 hardware 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
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
  - 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 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

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
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)
  - typically only 2~4, unlikely to be >10
  - 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
- “Higher-level” parallelism can be expressed across loop iterations
**Data Parallelism**

- Data Parallelism is abundant in many matrix operations and scientific/numerical applications
- Example: DAXPY/LINPACK (inner loop of Gaussian elimination and matrix-mult)
  ```c
  double Y[N], X[N];
  for (i=0; i<N; i++) {
    Y[i]=a*X[i]+Y[i]
  }
  ```
  - Y and X are vectors
  - same operations repeated on each Y[i] and X[i]
  - no data dependence across iterations

How would you map this to hardware?

---

**Data Parallel Execution**

```c
double A[N], B[N], C[N];
for (i=0; i<N; i++) {
  C[i]=foo(A[i], B[i])
}
```

- Assume foo() is a DFG like we seen earlier
- Instantiate \( k \) copies of the hardware unit foo to process \( k \) iterations of the loop in parallel
Pipeline Parallelism

- comb. logic for `foo` (delay=T)
  - throughput=\((1/T)\)
- \(T/2\) through \(T/2\)
  - throughput=\((2/T)\)
- \(T/3\) through \(T/3\) through \(T/3\)
  - t.p.=\((3/T)\)

Assume computing the same “function” repeatedly and independently.

Streaming Execution

```c
double A[N], B[N], C[N];
for (i=0; i<n; i++) {
    C[i]=foo(A[i], B[i])
}
```

- Build a deeply pipelined (high-frequency) version of `foo`()

Recall, pipeline works best when repeating identical and independent compute.
Not Always Straightforward

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

Loop Interchange

\[
\begin{align*}
&\text{for (k=0; k<N; k++)} \\
&\quad \text{for (i=0; i<N; i++)} \\
&\quad \quad \text{for (j=0; j<N; j++)} \\
&\quad \quad \quad C[i][j] = C[i][j] + A[i][k] \times B[k][j]
\end{align*}
\]
DRAM is Not Truly Random Access

- On DE4 FPGA platform w/ DDR2-800 SO-DIMM
- “packet” = aligned consecutive data
- transfer packets that are separated by large strides

Blocking for Data Reuse

```
for(i0=0; i0<N; i0+=B) {
    for(j0=0; j0<N; j0+=B) {
        for(k0=0; k0<N; k0+=B) {
            for(i=i0; i<i0+B; i++) {
                for(j=j0; j<j0+B; j++) {
                    for(k=k0; k<k0+B; k++) {
                        C[i][j]+=A[i][k]*B[k][j];
                    }
                }
            }
        }
    }
}
```

- Imagine a ‘N/B’x’N/B’ MATRIX of BxB matrix
  - inner-triple is straightforward matrix-matrix mult
  - outer-triple is MATRIX-MATRIX mult
  - for data-reuse keep BxB sub-matrices in on-chip SRAM
C-to-Hardware is for Real Today

- Many commercial and research tools are available
  - most major CAD vendors
  - Xilinx AutoESL and Altera OpenCL
  - ROCCC [UC Riverside] and LegUP [U Toronto] (free)
  - LLVM makes it pretty easy to roll-your-own
- State of technology
  - often specialized for particular usage or domain
  - work well on some domain or applications
  - not without blindspots

For more, see Special Issue on High-Level Synthesis, *IEEE Design & Test of Computers*, No. 4, Jul 2009

Promising Low Hanging Fruits

- Compiling control programs to FSMs instead of microcontrollers
- Mapping stylized code to stylized targets, e.g., loop nest to streaming
  - code structure gives strong hints to desired HW structure
  - specialized support and optimization
- Explicit parallelism, e.g. reinterpret OpenCL in HW

Either make my job simpler and/or my results better
Recap the Last Hour

• 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