18-643 Lecture 8: Abstractions for HW

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

• Your goal today: get a glimpse of classic high-level abstractions for hardware

• Notices
  – Handout #3: lab 1, due noon, 9/23
  – Handout #4: lab 2, due noon, 10/7

• Readings
  – Ch 5, Reconfigurable Computing
  – skim if interested: Ch 8, 9, 10, Reconfigurable Computing
Structural RTL

- Designer in charge
  - precise control at the bit and cycle granularity
  - arbitrary control and datapath schemes
  
  comes with the associated burdens

- RTL synthesis is literal
  - little room for optimizations (except comb. logic)
  - faithful to both “necessary” and “artifacts”
  
e.g., a and b mutually exclusive?

```vhdl
always@(posedge c)
  if (a)
    o<=1;
  else if (b)
    o<=2;
```

What is High-Level?

- Abstract away detail/control from designer
  - pro: **need not** spell out every detail
  - con: **cannot** spell out every detail

- Missing details?
  - implied in the abstraction, and/or
  - filled in by the synthesis tool

- To be meaningful
  - reduce work, and/or
  - improve outcome

  In HW practice, low tolerance for
degraded outcome regardless of ease
Today’s Topics

- Systolic Array
- Dataflow
- Streams
- Data Parallel (vector vs SIMD)
- Commonalities to look for
  - reduce complexity (how much has to be specified)
  - supports scalable parallelism under simplified global coordination
  - allows efficient hardware utilization
  - doesn’t work on every problem

C-to-HW next lecture . . .

Systolic Array

- An array of nodes (imagine each an FSM or FSM-D)
  - strictly, nodes are identical; cannot know the size of the array or position in the array
  - could generalize to other structured topologies
- Globally synchronized by “pulses”; on each pulse
  - exchange bounded data with direct neighbors
  - perform bounded compute on fixed local storage

  \( O(1) \) everything

- Simple
  - no external memory
  - no global interactions (except for the pulse)
E.g. Matrix-Matrix Multiplication

\begin{align*}
a &= \text{nan}; \\
b &= \text{nan}; \\
\text{accum} &= 0;
\end{align*}

For each pulse {
\begin{align*}
&\text{send-W}(a); \text{send-S}(b); \\
&a = \text{rcv-E}(); \\
&b = \text{rcv-N}(); \\
&\text{if } (a = \text{nan}) \\
&\quad \text{accum} = a * b + \text{accum}; \\
&}
\end{align*}

- Works for any \( N \)
- Only stores 3 vals per node
- If \( N > n \), emulate at \( N^2/n^2 \) slowdown

\begin{itemize}
  \item Parallel and scalable in nature
    \begin{itemize}
      \item can efficiently emulate key aspects of streams and data-parallel
      \item easy to build corresponding HW on VLSI (especially 1D and 2D arrays)
    \end{itemize}
  \item No global communication, except for pulse
  \item Scope of design/analysis/debug is 1 FSM-D
  \item Great when it works
    \begin{itemize}
      \item linear algebra, sorting, FFTs
      \item works more often than you think
      \item but clearly not a good fit for every problem
    \end{itemize}
\end{itemize}
Dataflow Graph

- Declarative representation (what, not how)
  - stateless
  - no imposed operation order, except by dependence
- Operate on flows (sequence of data values)
  i.e., \( X = \{ x_1, x_2, x_3, \ldots \} \), “1” = \{1,1,1,1, \ldots \}
- Flow operators, e.g., switch, merge, duplicate
- Temporal operators, e.g. \( \text{pre}(X) = \{\text{nil}, x_1, x_2, x_3, \ldots \} \)

\[ \begin{align*}
W &= X + 1 \\
X &= 2Y + Z \\
Y &= X
\end{align*} \]

What do you make of this?

node ACCUM(init, incr: int; reset: bool) returns (n: int);
let
  n = init -> if reset then init else pre(n) + incr
tel

\[ \begin{align*}
\text{pre}([e_1, e_2, e_3, \ldots]) &= \{\text{nil}, e_1, e_2, e_3, \ldots\} \\
\{e_1, e_2, e_3, \ldots\} &\rightarrow \{f_1, f_2, f_3, \ldots\} \]
E.g. Simulink Programming (RGB-to-Y)

[Figure 8.1: “Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation”]

Dataflow Take Away

- Naturally express fine-grain parallelism
  Many variations, asynchronous, dynamic, . . .
- Loose coupling between operators
  - synchronize by order in flow, not cycle or time
  - no imposed operation ordering
  - no global communications
- Declarative nature permits implementation flexibilities
- Great when it works
  - excellent match with signal processing
  - but clearly not a good fit for every problem
Stream Processing

- Similarity with static dataflow
  - operate on data in sequence (no random access)
  - repeat same operation on data in a stream
  - simple I/O (data source and sink)
- Differences
  - coarser operators
  - input and output flows not “synchronized”
  - operator can have a fixed amount of memory
    - buffer/compute over a window of values
    - carry dependencies over values in a stream

Streams Take Away

- Amenable to a high-degree of pipeline parallelism in between operators and within an operator
- No global synchronization or communication
- Good modularity
  - design in terms of composing valid stream-to-stream transformations
  - simple, elastic one-style stream “interface”
- Great when it works
  - excellent match with media processing, but also classic data mining, pattern discovery, and ML
  - but clearly not a good fit for every problem
Data Parallelism

• Abundant in 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])
}
```

• Instantiate \( k \) copies of the hardware unit \( \text{foo} \) to process \( k \) iterations of the loop in parallel
Pipelined Execution

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

E.g. SIMD Matrix-Vector Mult

// Each of the P threads is responsible for // M/P rows of A; self is thread id
for(i=self*M/P;i<((self+1)*M/P);i++) {
    y[i]=0;
    for(j=0;j<N;j++) {
        y[i]+=A[i][j]*x[j];
    }
}
E.g. Vectorized Matrix-Vector Mult

Repeat for each row of \( A \)

\[
\begin{align*}
LV & V1, Rx \quad ; \text{load vector } x \\
LV & V2, Ra \quad ; \text{load } i^{th} \text{ row of } A \\
\text{MULV} & V3,V2,V1 \quad ; \text{element-wise mult} \\
\text{"reduce"} & F0, V3 \quad ; \text{sum elements to scalar} \\
S.D & Ry, F0 \quad ; \text{store scalar result}
\end{align*}
\]

\[y = y + Ax\]

BTW, above is analogous to the SIMD code

E.g. Vectorized Matrix-Vector Mult

Repeat for each column of \( A \)

\[
\begin{align*}
\text{LVWS} & V0,(Ra,Rs) \quad ; \text{load-strided } i^{th} \text{ col of } A \\
\text{L.D} & F0,Rx \quad ; \text{load } i^{th} \text{ element of } x \\
\text{MULVS.D} & V1,V0,F0 \quad ; \text{vector-scalar mult} \\
\text{ADDV.D} &Vy,Vy,V1 \quad ; \text{element-wise add}
\end{align*}
\]

\[y = y + Ax\]
Data-Parallel Take Away

• Simplest but highly restricted parallelism
• Open to mixed implementation interpretations
  – SIMD parallelism +
  – (deep) pipeline parallelism
• Great when it works
  – important form of parallelism for scientific and numerical computing; made popular by GPGPUs
  – but clearly not a good fit for every problem

Altera and Xilinx support FPGA-accelerated OpenCL compute and OpenCL IP synthesis

Commonalities Revisited

• Parallelism under simplified global coordination
  – enforced regularity
  – asynchronous coupling
• Straightforward efficient mapping to hardware
  – low performance overhead
  – low resource overhead
  – high resource utilization
• Simplify design without interfering with quality
• But only works on specific problem patterns
Parting Thoughts: Conflict between High-Level and Generality

high-level: tools know better than you

RTL synthesis: general-purpose but special handling of structures like FSM, arith, etc.

place-and-route: works the same no matter what design

What about C for HW?

- Common arguments for using C to design HW
  - popularity
  - algorithm specification
- A large semantic gap to bridge
  - sequential thread of control
  - abstract time
  - abstract I/O model
  - functions only have a cost when executing
  - missing structural notions: bit width, ports, modules
- Still, no problem getting HW from C

How to get “good” hardware from C?
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);
}
```

Questions for Next Time

- 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?