18-643 Lecture 8: Abstractions for HW

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: survey classic high-level abstractions for hardware
- Notices
  - no office hours tomorrow (go to EGO Picnic)
  - Handout #3: lab 1, due noon, 9/22
  - Handout #4: lab 2, due noon, 10/6
- Readings
  - Ch 5, Reconfigurable Computing
  - skim if interested: Ch 8, 9, 10, Reconfigurable Computing
  - (for lab2) C. Zhang, et al., ISFPGA, 2015.
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?

```verilog
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 must be filled by someone
  - 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
What Models HW well?

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

**Function vs Execution vs Implementation**

--From Lecture Notes Devised by James C. Hoe, CMU/ECE/CALCM, ©2017--

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
    \[ O(1) \text{ everything} \]
- Simple
  - no external memory
  - no global interactions (except for the pulse)
**Example: Matrix-Matrix Multiplication**

```plaintext
a=nan;
b=nan;
accum=0;

For each pulse {
  send-W(a); send-S(b);
  a=rcv-E(); b=rcv-N();
  if (a!=nan)
    accum=a*b+accum;
}
```

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

---

**What comes to mind when you see?**

```plaintext
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];
    }
  }
}
```
**Systolic Array Take Away**

- Parallel and scalable in nature
  - can efficiently emulate key aspects of streams and data-parallel
  - easy to build corresponding HW on VLSI (especially 1D and 2D arrays)
- No global communication, except for pulse
- Scope of design/analysis/debug is 1 FSM-D
- Great when it works
  - linear algebra, sorting, FFTs
  - works more often than you think
  - 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)

\[
Y = a*X + Y = \begin{cases} 
\text{for}(i=0; \; i<N; \; i++) \{ \\
Y[i] = a*X[i] + Y[i] \\
Y[i] = a*X[i] + Y[i] \\
\} 
\end{cases}
\]

- \(Y\) 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

\[
\text{for}(i=0; i<N; i++) \{
    C[i] = \text{foo}(A[i], B[i])
\}
\]

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

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

Pipelined Execution

\[
\text{for}(i=0; i<N; i++) \{
    C[i] = \text{foo}(A[i], B[i])
\}
\]

- Build a deeply pipelined (high-frequency) version of \(\text{foo}()\)

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

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];
    }
}

This is bad news if A is column-major

### E.g. Vectorized Matrix-Vector Mult

Repeat for each row of A:

- LV V1, Rx ; load vector x
- LV V2, Ra ; load i’th row of A
- MULV V3,V2,V1 ; element-wise mult
- “reduce” F0, V3 ; sum elements to scalar
- S.D Ry, F0 ; store scalar result

This is fine if A is row-major.
**E.g. Vectorized Matrix-Vector Mult**

Repeat for each column of $A$

- `LVWS V0,(Ra,Rs)`; load-strided $i$’th col of $A$
- `L.D F0,Rx`; load $i$’th element of $x$
- `MULVS.D V1,V0,F0`; vector-scalar mult
- `ADDV.D Vy,Vy,V1`; element-wise add

BTW, above is analogous to the SIMD code

**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
  - but clearly not a good fit for every problem
Dataflow Graphs

- Consider a von Neumann program
  - what is the significance of the program order?
  - what is the significance of the storage locations?

\[
\begin{align*}
v & := a + b; \\
w & := b * 2; \\
x & := v - w \\
y & := v + w \\
z & := x * y
\end{align*}
\]

- Dataflow operation ordering and timing implied in data dependence
  - instruction specifies who receives the result
  - operation executes when all operands received
  - “source” vs “intermediate” representation

Token Passing

- fan-in
- fan-out
- switch (conditional)
- merge (conditional)

"fire" output tokens when all required input present

Consider multi-, variable-cycle ops and links
Synchronous Dataflow

- 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. \( \text{pre}(X) = \{\text{nil}, x_1, x_2, x_3, \ldots \} \)

Fig 1, Halbwachs, et al., The Synchronous Data Flow Programming Language LUSTRE

Function vs Execution vs Implementation

What do you make of this?

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

\( \text{pre}(\{e_1, e_2, e_3, \ldots \}) = \{\text{nil}, e_1, e_2, e_3, \ldots \} \)
\( \{e_1, e_2, e_3, \ldots \} \rightarrow \{f_1, f_2, f_3, \ldots \} = \{e_1, f_2, f_3, f_4, \ldots \} \)
E.g. Simulink Programming (RGB-to-Y)

Dataflow Take Away

- Naturally express fine-grain, implicit 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 dataflow
  - operate on data in sequence (no random access)
  - repeat same operation on data in a stream
  - simple I/O (data source and sink)
- More flexible rules
  - coarser operators
  - input and output flows need not be synchronized or rate-matched
  - 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
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
- abstract

- special handling of structures like FSM, arith, etc.
- place-and-route: works the same no matter what design
- General

- nuts & bolts
- RTL synthesis: general-purpose
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?