18-643 Lecture 8:
High-Level Abstractions for HW

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: survey classic high-level abstractions for hardware

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

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

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

- Designer in charge
  - arbitrary control and datapath schemes
  - precise control—when, what where—at the bit and cycle granularity

With great power comes great responsibility . . .
Crux of the RTL Design Difficulty

- We design FSM-Ds separately
  - liable to forget what one machine is doing when focusing on another
- No language support for coordination
  - no explicit way to say how state transitions of two FSMs must be related
- Coordination hardcoded into design implicitly
  - leave little room for automatic optimization
  - hard to localize design changes
  - (unless decoupled using request/reply-style handshakes)
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
- Dataflow
- Streams
- Commonalities to look for
  - reduce design complexity/effort
  - supports scalable parallelism under simplified global coordination (by imposing a “structure”)
  - allows straightforward, efficient HW mapping
  - doesn’t work on every problem
Systolic Array

• An array of PEs (imagine each an FSM or FSM-D)
  – strictly, PEs 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

  \[ \text{O}(1) \text{ everything} \]

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

- Works for any $n$
- Only stores 3 vals per PE
- If $N>n$, emulate at $N^2/n^2$ slowdown
Does the last slide come to mind when you see??

```c
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)

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

- \( Y \) and \( X \) are vectors
- \( \sqrt{ } \) same operations repeated on each \( Y[i] \) and \( X[i] \)
- \( \sqrt{ } \) no data dependence across iterations

*How would you exploit this in hardware?*
Data Parallel Execution

```c
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

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

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

How to structure memory and array layout?
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

no such instruction
E.g. Vectorized Matrix-Vector Mult

Repeat for each column of A

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

Above is analogous (when/what/where) 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?

• 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

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

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**

$$Y \rightarrow X$$
$$Z \rightarrow X$$

$$2 \rightarrow X$$
$$1 \rightarrow +$$
$$W = X + 1$$

$$X = 2Y + Z$$

---

Function vs Execution vs Implementation
What do you make of this?

node ACCUM(init, incr: int; reset: bool) returns (n: int);

let

\[
\begin{align*}
n &= \text{init} \rightarrow \text{if reset then init else pre}(n) + \text{incr} \\
\end{align*}
\]

tel

\[
\begin{align*}
\text{pre}\{e_1, e_2, e_3, \ldots\} &= \{\text{nil, } e_1, e_2, e_3, \ldots\} \\
\{e_1, e_2, e_3, \ldots\} + \{f_1, f_2, f_3, \ldots\} &= \{e_1, f_2, f_3, f_4, \ldots\}
\end{align*}
\]
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, 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)

• Emphasis on IPs and their composition

• 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
E.g., Vision Processing Pipeline

Color-based object tracking (linear pipeline, 4 stages)

1. Gaussian blur  
2. Color threshold  
3. Color threshold  
4. Color threshold  
Display

Background subtraction (2-branch pipeline, 6 stages)

1. Duplicate  
2. Gaussian blur  
3. Background subtraction  
4. Synchronizer  
5. Merge  
6. Paint  
Display

Corner + edge detection (3-branch pipeline, 10 stages)

1. Duplicate  
2. Duplicate  
3. Corner detection  
4. Synchronizer  
8. Merge  
9. Merge  
10. Paint  
Display
E.g., Network Packet Processing

- Ethernet (ethrnt)
- TCP flow reassembly
- "fast pattern" matching
- 2nd filtering
- Offloading to CPU
- CPU full matching

https://github.com/cmu-snap/pigasus
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

insist on quality
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?