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
  – 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
Course Project Template

• Pick a “compute” application; off-chip data
• Pick a metric of merit
• **Study** implementation options
  – a good software implementation must be one option
  – the rest is up to you
• Report findings

  Keep in mind, you have optimistically 6 weeks; don’ forget you are taking other courses
Proposal

• PPT slides instead of write-up
  – introduction (what is the problem and metric)
  – motivation (why is it important/interesting)
  – overview of approach (what do you do exactly)
  – overview of expected results
  – weekly “testable” milestones
  – risk assessment (what are the unknowns)
    • do you have everything you need to start?
    • do you know how to do every step?
    • do you know already what will happen?
• 15 minutes to present subset to class for feedback
Advanced Warnings

• Rolling start
  – in-class proposal presentation (worth 25%) on wk 9
  – midterm on week 8; Lab 3 on week 7&8

• Peer grading of proposal
  – your peers will evaluate you too
  – 10% them; 15% me

• Convince/impress us
  – it is worth doing
  – 6 wks worth of work (for a 12-unit load for 2 or 3)
  – reasonable chance of meeting milestones
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;
```
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 (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
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

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

```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;
}
```
What comes 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
− 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

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

- Instantiate $k$ copies of the hardware unit $foo$ to process $k$ iterations of the loop in parallel
Pipelined Execution

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

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

![Diagram](image)

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

$y = \begin{bmatrix} M \\ N \end{bmatrix} A \begin{bmatrix} M \\ N \end{bmatrix}$
E.g. Vectorized Matrix-Vector Mult

Repeat for each column of A

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

BTW, above is analogous to the SIMD code

\[
\begin{align*}
\begin{bmatrix}
Y
\end{bmatrix} &= \\
& \begin{bmatrix}
A
\end{bmatrix} \\
& \begin{bmatrix}
X
\end{bmatrix}
\end{align*}
\]

\[
\begin{array}{c}
\text{M} \\
\text{M} \\
\text{N} \\
\text{N}
\end{array}
\]
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?

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

• 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

[figure and example from Arvind]
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. $\text{pre}(X)=$\{nil, x1, x2, x3, \ldots\}

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

Function vs Execution vs Implementation
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

pre({e₁, e₂, e₃, ....}) is {nil, e₁, e₂, e₃, ....}
{e₁, e₂, e₃, ....}->{f₁, f₂, f₃, ....} is {e₁, f₂, f₃, f₄ ....}
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)
• 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

insist on quality

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

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

high-level: tools know better than you

nuts&bolts

abstract

specialized
general

18-643-F19-L08-S30, James C. Hoe, CMU/ECE/CALCM, ©2019
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?
DoF: you pick the application

- The problem could be
  - well studied (expect thoroughness and depth)
  - unproven (credit for honest attempts)

  Absolute yardstick: is it 6 weeks of effort

- Something there is a reason to do on FPGAs
- Best if it is something you want to or have to do anyways

- Need to find and study (at least) 1 closely relevant research paper as starting point
DoF: you define the metric

• What you can study
  – performance (throughput or latency?)
  – cost (in terms of what?)
  – power and energy (how will you measure?)
  – design effort (what will you measure)
  – app-specific metrics (e.g., numerical accuracy)
  – composite metric: energy-delay-product, performance/watt, performance/$

• Must commit up-front
  – measurement procedure/benchmark
  – testable “good-enough” target condition
DoF: Platform

• You have the Ultra96
• You may substitute a reconfigurable platform you are already using (check with me first)
• You have access to more advanced platforms
  – risky learning curve to fit in 6 weeks
  – only if this plays into what else you are doing in life
DoF: Approach

• 1 option must be a good software-only baseline
• This is a “study”
  – do more than crank out implementations
  – think about what are the design choices
  – hypothesize the expected effects of your choices
  – corroborate hypothesis by implementation and evaluation
• Implementation approach:
  – no artificial bounds
  – how would you work in real-life?
  – if you have access, you can use it (including tools and IPs)

Convince us it is 6 weeks of effort