ESE535: Electronic Design Automation

Day 14: March 11, 2013
C → RTL

Today
See how get from a language (C) to dataflow
• Basic translation
  – Straight-line code
  – Memory
  – Basic Blocks
  – Control Flow
  – Looping
• Optimization
  – If-conversion
  – Hyperblocks
  – Common Optimizations
  – Pipelining
  – Unrolling

Behavioral (C, MATLAB, …)
Arch. Select Schedule
RTL
FSM assign
Two-level, Multilevel opt.
Covering
Retiming
Gate Netlist
Placement
Routing
Layout
Masks

Day 1
Design Productivity by Approach

C Primitives
Arithmetic Operators
• Unary Minus (Negation) -a
• Addition (Sum) a + b
• Subtraction (Difference) a - b
• Multiplication (Product) a * b
• Division (Quotient) a / b
• Modulus (Remainder) a % b

C Primitives
Bitwise Operators
• Bitwise Left Shift a << b
• Bitwise Right Shift a >> b
• Bitwise One’s Complement ~a
• Bitwise AND a & b
• Bitwise OR a | b
• Bitwise XOR a ^ b

C Primitives
Comparison Operators
• Less Than a < b
• Less Than or Equal To a <= b
• Greater Than a > b
• Greater Than or Equal To a >= b
• Not Equal To a != b
• Equal To a == b
• Logical Negation !a
• Logical AND a && b
• Logical OR a || b

Source: Keutzer (UCB EE 244)
Expressions: combine operators

- \( a \times x + b \)

A connected set of operators → Graph of operators

C Assignment

- Basic assignment statement is: Location = expression
- \( f = a \times x + b \)

Straight-line code

- A sequence of assignments
  - What does this mean?
  - \( g = a \times x; \)
  - \( h = b + g; \)
  - \( i = h \times x; \)
  - \( j = i + c; \)

Variable Reuse

- Variables (locations) define flow between computations
- Locations (variables) are reusable
  - \( t = a \times x; \)
  - \( r = t \times x; \)
  - \( t = b \times x; \)
  - \( r = r + t; \)
  - \( r = r + c; \)

Variable Reuse

- Variables (locations) define flow between computations
- Locations (variables) are reusable
  - \( t = a \times x; \)
  - \( r = t \times x; \)
  - \( t = b \times x; \)
  - \( r = r + t; \)
  - \( r = r + c; \)
- Sequential assignment semantics tell us which definition goes with which use.
  - Use gets most recent preceding definition.
Dataflow

• Can turn sequential assignments into dataflow graph through def→use connections
  \[ t = a \times x; \quad r = t \times x; \]
  \[ t = b \times x; \quad r = r + t; \]
  \[ r = r + c; \]

Dataflow Height

• Height (delay) of DF graph may be less than # sequential instructions.

Lecture Checkpoint

• Happy with
  – Straight-line code
  – Variables

• Next topic: Memory

C Memory Model

• One big linear address space of locations
• Most recent definition to location is value
• Sequential flow of statements

C Memory Operations

Read/Use
• \( a = *p; \)
• \( a = p[0]; \)
• \( a = p[c*10+d]; \)

Write/Def
• \( *p = 2*a + b; \)
• \( p[0] = 23; \)
• \( p[c*10+d] = a \times x + b; \)

Memory Operation Challenge

• Memory is just a set of location
• But memory expressions can refer to variable locations
  – Does \(*q\) and \(*p\) refer to same location?
  – \(*p\) and \(q[c*10+d]\)?
  – \(p[0]\) and \(p[c*10+d]\)?
  – \(p[f(a)]\) and \(p[g(b)]\)?
Pitfall

- $P[i]=23$
- $r=10+P[i]$
- $P[j]=17$
- $s=P[j]*12$

- Could do:
  - $P[i]=23; P[j]=17; r=10+P[i]; s=P[j]*12$

- Value of $r$ and $s$? 
  - ...unless $i=j$
  - Value of $r$ and $s$?

C Pointer Pitfalls

- *$p=23$
- $r=10+*p;$
- *$q=17$
- $s=*q*12;$

- Similar limit if $p==q$

C Memory/Pointer Sequentialization

- Must preserve ordering of memory operations
- A read cannot be moved before write to memory which may redefine the location of the read
  - Conservative: any write to memory
  - Sophisticated analysis may allow us to prove independence of read and write
- Writes which may redefine the same location cannot be reordered

Consequence

- Expressions and operations through variables (whose address is never taken) can be executed at any time
  - Just preserve the dataflow
- Memory assignments must execute in strict order
  - Ideally: partial order
  - Conservatively: strict sequential order of C

Forcing Sequencing

- Demands we introduce some discipline for deciding when operations occur
  - Could be a FSM
  - Could be an explicit dataflow token
  - Callahan uses control register
- Other uses for timing control
  - Control
  - Variable delay blocks
  - Looping

Scheduled Memory Operations

Source: Callahan
Control

Conditions

- If (cond) – DoA
- Else – DoB
- While (cond) – DoBody
- No longer straightline code
- Code selectively executed
- Data determines which computation to perform

Basic Blocks

- Sequence of operations with
  - Single entry point
  - Once enter execute all operations in block
  - Set of exits at end

- Can dataflow schedule operations within a basic block
  - As long as preserve memory ordering

Connecting Basic Blocks

- Connect up basic blocks by routing control flow token

  - May enter from several places
  - May leave to one of several places

  A=B+C   BB0: B=B+C   BB1:
  E=A*D   A=B+C   Q++
  If (E>100) E=A*D   E=E-100
  {     t=(E>100)     br bb2
    Q++;     bb(t,bb1,bb2)     BB2:
    E=E-100;           G=F*E
  } G=F*E;

Basic Blocks

- Sequence of operations with
  - Single entry point
  - Once enter execute all operations in block
  - Set of exits at end

- Connect up basic blocks by routing control flow token

  - May enter from several places
  - May leave to one of several places

  A=B+C   BB0: B=B+C   BB1:
  E=A*D   A=B+C   Q++
  If (E>100) E=A*D   E=E-100
  {     t=(E>100)     br bb2
    Q++;     bb(t,bb1,bb2)     BB2:
    E=E-100;           G=F*E
  } G=F*E;

Connecting Basic Blocks
Basic Blocks for if/then/else

Loops

sum=0;
for (i=0; i<imax; i++)
    sum+=i;
    r=sum<<2;

Lecture Checkpoint

- Happy with
  - Straight-line code
  - Variables
  - Memory
  - Control

- Q: Satisfied with implementation this is producing?

Beyond Basic Blocks

- Basic blocks tend to be limiting
- Runs of straight-line code are not long
- For good hardware implementation
  - Want more parallelism

Simple Control Flow

If (cond) {...} else {...}

Assignments become conditional
- In simplest cases (no memory ops), can treat as dataflow node

Simple Conditionals

if (a>b)
c=b*c;
else
c=a*c;
**Simple Conditionals**

\[
v = a; \\
\text{if } (b > a) \\
v = b;
\]

- If not assigned, value flows from before assignment

**Recall: Basic Blocks for if/then/else**

- Mux converted version has shorter path (lower latency)
- Why?

**Height Reduction**

- Mux converted version has shorter path (lower latency)
- Can execute condition in parallel with then and else clauses
Mux Conversion and Memory

- What might go wrong if we mux-converted the following:
  - If (cond)
    - *a=0
  - Else
    - *b=0

Don’t want memory operations in non-taken branch to occur.

Conclude: cannot mux-convert blocks with branches (without additional care).

Hyperblocks

- Can convert if/then/else into dataflow
  - If/mux-conversion
- Hyperblock
  - Single entry point
  - No internal branches
  - Internal control flow provided by mux conversion
  - May exit at multiple points

Basic Blocks → Hyperblock

Source: Callahan

Hyperblock Benefits

- More code → typically more parallelism
  - Shorter critical path
- Optimization opportunities
  - Reduce work in common flow path
  - Move logic for uncommon case out of path
    - Makes smaller faster
Optimizations

- Constant propagation: \( a=10; \) \( b=c[a]; \)
- Copy propagation: \( a=b; \) \( c=a+d; \)
- Constant folding: \( c[10*10+4]; \) \( \rightarrow c[104]; \)
- Identity Simplification: \( c=1^*a+0; \) \( \rightarrow c=a; \)
- Strength Reduction: \( c=b^*2; \) \( \rightarrow c=b<<1; \)
- Dead code elimination
- Common Subexpression Elimination:
  \( C[x*100+y]=A[x*100+y]+B[x*100+y] \)
  \( \rightarrow t=x*100+y; \) \( C[t]=A[t]+B[t]; \)
- Operator sizing:

Additional Concerns?

What are we still not satisfied with?

- Parallelism in hyperblock
  - Especially if memory sequentialized
  - Disambiguate memories?
  - Allow multiple memory banks?
- Only one hyperblock active at a time
  - Share hardware between blocks?
- Data only used from one side of mux
  - Share hardware between sides?
- Most logic in hyperblock idle?
  - Couldn’t we pipeline execution?

Pipelining

for \((i=0;i<\text{MAX};i++)\)
\( o[i]=(a*x[i]+b)*x[i]+c; \)
- If know memory operations independent

Unrolling

for \((i=0;i<\text{MAX};i++)\)
\( o[i]=(a*x[i]+b)*x[i]+c; \)
for \((i=0;i<\text{MAX};i+=2)\)
\( o[i]=(a*x[i]+b)*x[i]+c; \)
\( o[i+1]=(a*x[i+1]+b)*x[i+1]+c; \)
Unrolling

• If MAX=4:
  \[ o[0] = (a \cdot x[0] + b) \cdot x[0] + c; \]
  \[ o[1] = (a \cdot x[1] + b) \cdot x[1] + c; \]
  \[ o[2] = (a \cdot x[2] + b) \cdot x[2] + c; \]
  \[ o[3] = (a \cdot x[3] + b) \cdot x[3] + c; \]
  for (i=0;i<MAX;i++)

  \[ o[i] = (a \cdot x[i] + b) \cdot x[i] + c; \]

  \[ o[i+1] = (a \cdot x[i+1] + b) \cdot x[i+1] + c; \]

Benefits?

Create larger basic block.
More scheduling freedom.
More parallelism.

Flow Review

Summary

• Language (here C) defines meaning of operations
• Dataflow connection of computations
• Sequential precedents constraints to preserve
• Create basic blocks
• Link together
• Optimize
  – Merge into hyperblocks with if-conversion
  – Pipeline, unroll
• Result is dataflow graph
  – (can schedule to RTL)

Big Ideas:

• Semantics
• Dataflow
• Mux-conversion
• Specialization
• Common-case optimization
Admin

• Assignment 5 out today
• Assignments 3 graded
• Reading for Wednesday online
• Office hour tomorrow (Tuesday)