ESE535:
Electronic Design Automation

Day 16: March 25, 2015
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

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

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

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

Things might have a hardware operator for…

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

Things might have a hardware operator for…

Domain-Specific RTL

Behavioral
Gate
Schedule
Placement
Routing
Layout
Masks

Design Productivity by Approach

Source: Keutzer (UCB EE 244)

GATES/WEEK

Domain Specific
Behavioral
RTL
Gate
Transistor

8K - 12K
2K - 10K
1K - 2K
100 - 200
10 - 20

100 - 200
1K - 10K
2K - 10K
8K - 12K

Source: Dataquest

C Primitives

Bitwise Operators

Things might have a hardware operator for…
Expressions:
combine operators

• \(a^2 + b\)

A connected set of operators
\(\rightarrow\) Graph of operators

Expressions:
combine operators

• \(a^2 + b\)
• \((a^2 + b) \times c\)
• \((a + 10) \times b < 100\)

A connected set of operators
\(\rightarrow\) Graph of operators

C Assignment

• Basic assignment statement is:
  Location = expression

• \(f = a^2 + b\)

Straight-line code

• a sequence of assignments
  • What does this mean?

  \[\begin{align*}
  g &= a^2 + x; \\
  h &= b + g; \\
  i &= h^2; \\
  j &= i + c;
  \end{align*}\]

Variable Reuse

• Variables (locations) define flow between computations
• Locations (variables) are reusable
  \[\begin{align*}
  t &= a^2 + x; \\
  r &= t^2; \\
  t &= b^2 + x; \\
  r &= r + t; \\
  r &= r + c;
  \end{align*}\]

• 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 \( \text{def} \rightarrow \text{use} \) connections

\[
\begin{align*}
t &= a \times x; \\
r &= t \times x; \\
t &= b \times x; \\
r &= r + t; \\
r &= r + c; \\
r &= r + c;
\end{align*}
\]

Dataflow Height

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

\[
\begin{align*}
t &= a \times x; \\
r &= t \times x; \\
t &= b \times x; \\
r &= r + t; \\
r &= r + c; \\
r &= r + c;
\end{align*}
\]

Lecture Checkpoint

- Happy with
  - Straight-line code
  - Variables
- Graph for preclass \( f \)
- 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
- \( *p = 2*a + b; \)
- \( p[0] = 23; \)
- \( p[c*10+d] = a^*x + b; \)

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?
  - \( p[0] \) and \( p[c*10+d] \)?
  - \( *p \) and \( q[c*10+d] \)?
  - \( p[f(a)] \) and \( p[g(b)] \)?
Pitfall

- \( P[i] = 23 \)
- \( r = 10 + P[i] \)
- \( P[j] = 17 \)
- \( s = P[j] \times 12 \)

- Value of \( r \) and \( s \)?

C Pointer Pitfalls

- \( *p = 23 \)
- \( r = 10 + *p \)
- \( *q = 17 \)
- \( s = *q \times 12 \)

- 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

begin:
  x:=y;
y++;;
z:=y;
t:=z>=20;
brfalse t, finish
y=4
finish:
x:=x*y;
end:

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

Connecting Basic Blocks

- Connect up basic blocks by routing control flow token
  - May enter from several places
  - May leave to one of several places
Basic Blocks for if/then/else

Loops

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

sum=0;
i=0;
while (i<imax)
    sum+=i;
i=i+1;
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;
if (b>a)
v=b;

• If not assigned, value flows from before assignment

Preclass G

• Finish drawing graph for preclass g

Mux Converted

if (a>10)
a++;
else;
a—
x=a^0x07

Recall: Basic Blocks for if/then/else

Source: Callahan

Height Reduction

• 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

Common Case Height Reduction

Source: Callahan

Common-Case Flow Optimization

Source: Callahan

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?

Optimizations

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

Pipelining

for (i=0; i<MAX; i++)
o[i]=(a*x[i]+b)*x[i]+c;

- If know memory operations independent
Unrolling

• Put several (all?) executions of loop into straight-line code in the body.

```
for (i=0;i<MAX;i++)
o[i]=(a*x[i]+b)*x[i]+c;
```

```
for (i=0;i<MAX;i+=2)
o[i]=(a*x[i]+b)*x[i]+c;
o[i+1]=(a*x[i+1]+b)*x[i+1]+c;
```

Benefits?

Unrolling

• If MAX=4:
  
  o[0]=(a*x[0]+b)*x[0]+c;
o[1]=(a*x[1]+b)*x[1]+c;

  for (i=0;i<MAX;i++)
o[i]=(a*x[i]+b)*x[i]+c;
```

```
for (i=0;i<MAX;i+=2)
o[i]=(a*x[i]+b)*x[i]+c;
o[i+1]=(a*x[i+1]+b)*x[i+1]+c;
```

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

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

• Project Assignment
• HW8
• Reading for Monday on web