ESE535: Electronic Design Automation

Day 15: March 23, 2008
C→RTL

Today

• Straight-line Code
• If-conversion
• Memory
• Basic Blocks and Control Flow
• Looping
• Hyperblocks
• Common Optimizations

So far…

• We’ve looked at RTL down
  – Start with:
    • Boolean logic equations
    • Registers
  – Retiming, FSM encoding, logic mapping, covering, placement

Raise Abstraction

• Want to start designs at higher level
  – VHDL (could still be RTL)
  – C

Day 1 Design Productivity by Approach

Today

• See how get from a language (C) to RTL level

Source: Keutzer (UCB EE 244)
Arithmetic Operators

- Unary Minus (Negation) \(-a\)
- Addition (Sum) \(a + b\)
- Subtraction (Difference) \(a - b\)
- Multiplication (Product) \(a \times b\)
- Division (Quotient) \(a / b\)
- Modulus (Remainder) \(a \% b\)

Bitwise Operators

- Bitwise Left Shift \(a \ll b\)
- Bitwise Right Shift \(a \gg b\)
- Bitwise One’s Complement \(\sim a\)
- Bitwise AND \(a \& b\)
- Bitwise OR \(a | b\)
- Bitwise XOR \(a ^ b\)

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 \(a \| \| b\)

Build complex expressions

- \(a \times x + b \times x + c\)
- \((a + 10) \times b < 100\)

C Assignment

- Basic assignment statement
- Location = expression
- \(F = a \times x + b \times x + c\)

Straight-line code

- Just 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

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

Variable Reuse

- Variables (locations) define flow between computations
- Locations (variables) are reusable

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

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 def→use connections

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

Simple Control Flow

- If (cond) { … } else { … }

- Assignments become conditional
- In simplest cases, can treat as dataflow node

Simple Conditionals

\[
\begin{align*}
\text{if (a>b)} & \quad \text{c = b } \times c; \\
\text{else} & \quad \text{c = a } \times c;
\end{align*}
\]

Simple Conditionals

\[
\begin{align*}
v &= a; \\
\text{if (b>a)} & \quad v = b;
\end{align*}
\]

- If not assigned, value flows from before assignment
**Simple Conditionals**

```plaintext
max = a;
min = a;
if (a > b)
    min = b;
c = 1;
else
    max = b;
c = 0;
```

- May (re)define many values on each branch.

**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 just a location
- But memory expressions can refer to variable locations
  - Does `*q` and `*p` refer to same location?
  - `*p` and `p[c*10+d]`?
  - `p[0]` and `p[c*10+d]`?
  - `p[f(a)]` and `p[q(b)]`?

**Pitfall**

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

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

- ...unless `i = j`

**C Pointer Pitfalls**

- `*p = 23`
- `*q = 17`
- `r = 10 + *p;`
- `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 (who 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
  – Variable delay blocks
  – Looping
  – Complex control

Scheduled Memory Operations
Source: Callahan

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
Basic Blocks for if/then/else

Loops

\[
\text{sum}=0; \\
\text{for } (i = 0; i < \text{imax}; i++) \\
\quad \text{sum} += i; \\
\quad r = \text{sum} \ll 2;
\]

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

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 $\rightarrow$ Hyperblock

Hyperblock Benefits
- More code $\rightarrow$ 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

Optimizations

- Constant propagation: \(a=10; b=[a]\);
- Copy propagation: \(a=b; c=a+d; \rightarrow c=b+d\);
- 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:
  - \(C[x*100+y]=A[x*100+y]+B[x*100+y]\)
  - \(t=x*100+y; C[t]=A[t]+B[t]\);
- Operator sizing: \(\text{for } (i=0; i<100; i++) b[i]=(a&0xff+i)\);

Flow Review

Concerns

- 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

\text{for } (i=0; i<\text{MAX}; i++)
\nu[i]=[(a\times[i]+b)]\times[i]+c;

- If know memory operations independent
Summary

- Language (here C) defines meaning of operations
- Dataflow connection of computations
- Sequential precedents constraints to preserve
- Create basic blocks
- Link together
- Merge into hyperblocks with if-conversion
- Result is logic and registers → RTL

Big Ideas:

- Dataflow
- Mux-conversion
- Specialization
- Common-case optimization

Admin

- Reading for Wednesday
- Assignment 5 out