ESE532: System-on-a-Chip Architecture

Day 10: February 15, 2017
High Level Synthesis (HLS)
C-to-gates

Today

- Spatial Computations from C specification
  - Basic transforms
  - Limitations from C semantics

Message

- C (or any programming language) specifies a computation
- Can describe spatial computation
- Underlying semantics is sequential
  - Watch for unintended sequentialization
  - Probably write C for spatial differently than you write C for processors

Coding Accelerators

- Want to exploit FPGA logic on Zynq to accelerate computations
- Traditionally has meant develop accelerators in
  - Hardware Description Language (HDL)
    - E.g. Verilog → undergrads see in CIS371
  - Directly in schematics
  - Generator language (constructs logic)

Course "Hypothesis"

- C-to-gates synthesis mature enough to use to specify hardware
  - Leverage fact everyone knows C
    - (must, at least, know C to develop embedded code)
  - Avoid taking time to teach Verilog or VHDL
    - Or making Verilog a pre-req.
  - Focus on teaching how to craft hardware
    - Using the C already know
    - …may require thinking about the C differently

Discussion [open]

- Is it obvious we can write C to describe hardware?
- What parts of C translate naturally to hardware?
- What parts of C might be problematic?
- What parts of hardware design might be hard to describe in C?

Advantage

- Use C for hardware and software
  - Test out functionality entirely in software
    - Debug code before put on hardware where harder to observe what's happening
  - Explore hardware/software tradeoffs by targeting same code to either hardware or software

Preclass F

- Ready for preclass f?
- Skip to preclass f

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

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…

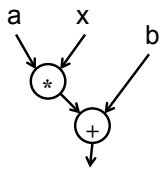
C Primitives Comparison Operators

- 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

Things might have a hardware operator for…

Expressions: combine operators

- a*x+b



A connected set of operators
→ Graph of operators

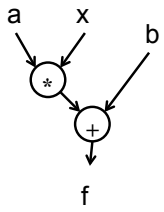
Expressions: combine operators

- a*x+b
- a*x*x+b*x+c
- a*(x+b)*x+c
- ((a+10)*b < 100)

A connected set of operators
→ Graph of operators

C Assignment

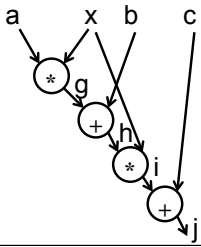
- Basic assignment statement is:
  Location = expression
- f=a*x+b



Straight-line code

- a sequence of assignments
- What does this mean?

```
g=a*x;
h=b+g;
i=h*x;
j=i+c;
```



Variable Reuse

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

```
t=a*x;
r=t*x;
t=b*x;
r=r+t;
r=r+c;
```


Variable Reuse

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

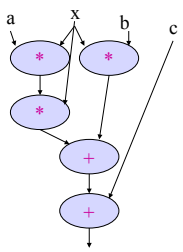
```
t=a*x;   t=a*x;
r=t*x;   r=t*x;
t=b*x;           t=b*x;
r=r+t;     r=r+t;
r=r+c;     r=r+c;
```

- Sequential assignment semantics tell us which definition goes with which use.
  - **Use** gets most recent preceding **definition**.

Dataflow

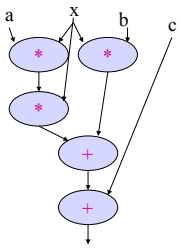
- Can turn sequential assignments into dataflow graph through def→use connections

```
t=a*x;   t=a*x;
r=t*x;   r=t*x;
t=b*x;           t=b*x;
r=r+t;     r=r+t;
r=r+c;     r=r+c;
```



Dataflow Height

- t=a*x; t=a*x;
  r=t*x; r=t*x;
  t=b*x; t=b*x;
  r=r+t; r=r+t;
  r=r+c; r=r+c;
- Height (delay) of DF graph may be less than # sequential instructions.

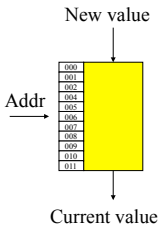


Lecture Checkpoint

- 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]*12

- Value of r and s?

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

….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

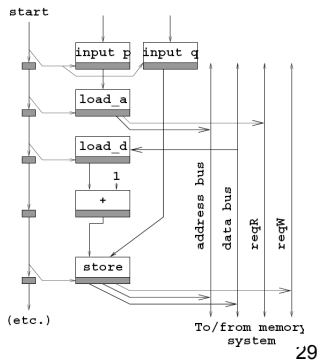
- **Expressions and operations** through variables (whose address is never taken) can be executed at any time
  - Just preserve the dataflow
- **Memory assignments** must execute in strict order
  - Ideally: partial order
  - Conservatively: strict sequential order of C

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

```
*q = *p + 1;
(etc.)
```



Source: Callahan

Hardware/Parallelism Challenge

- Can we give enough information to the compiler to
  - allow it to reorder?
  - allow to put in separate embedded memories?
- Is the compiler smart enough to exploit?

Multiple Memories

- How might we want this to be implemented using multiple memories?

```
for(i=0;i<MAX;i++)
    C[i]=A[i]*B[i];
```


Idioms

Hard?

```
void fun(int *a, int *b, int *c)
for(i=0;i<MAX;i++)
    A[i]=A[f(i)];
```

– Data-dependent relationship

Easier

```
int a[1024], b[1024],
    c[1024];
for (i=0;i<MAX;i++)
    A[2*i+3]=A[i]+A[i+2];
```

– Linear equations, can potentially solve for relationship

Memory Allocation?

- How support malloc() in hardware?

Hardware Memory

- Typically small, fixed, local memory blocks
- Reuse memory blocks
  - Not allocate new blocks
  - Cannot make data-dependent memory sized blocks
- Different hardware units → different local memories
  - move data not pass pointers

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:
```

```
BB0:
  x=y;
  y++;
  z=y;
  t=z>20
  br(t,BB1,BB2)
```

```
BB1:
  y=4;
  br BB2
```

```
BB2:
  x=x*y;
```

Basic Blocks?

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

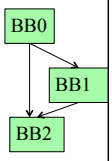
- Connect up basic blocks by routing control flow token
  - May enter from several places
  - May leave to one of several places

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

```
BB0:
  x=y;
  y++;
  z=y;
  t=z>20
  br(t,BB1,BB2)
```

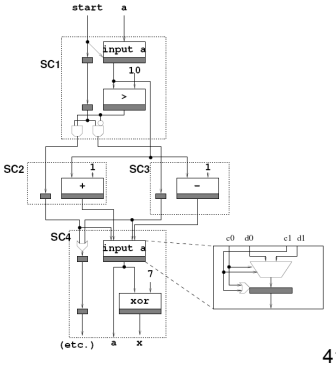
```
BB1:
  y=4;
  br BB2
```

```
BB2:
  x=x*y;
```



Basic Blocks for if/then/else

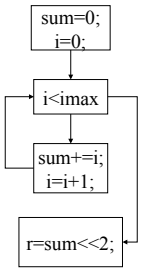
```
if (a>10) {
  a++;
} else {
  a--;
}
x = a ^ 7;
(etc.)
```



Source: Callahan

Loops

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



Lecture Checkpoint

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

Function Call

- What do we do with function calls?

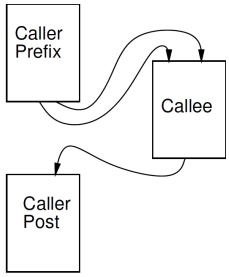
Inline

```
int f(int a, int b)
    return(sqrt(a*a+b*b));

for(i=0;i<MAX;i++)
    D[i]=f(A[i],B[i]);
```

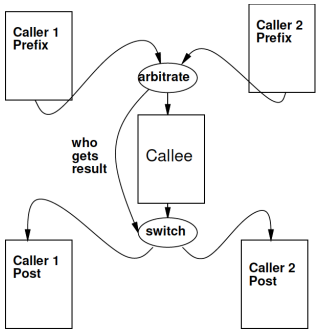
- for(i=0;i<MAX;i++)
  D[i]=sqrt(A[i]*A[i]+B[i]*B[i]);

Treat as data flow

- Implement function as an operation
- Send arguments as input tokens
- Get result back as token



Shared Function



Recursion?

```
int fib(int x) {
  if ((x==0) || (x==1))
  return(1);
 else
  return(fib(x-1)+fib(x-
  2));
}
```

- In general won't work.
  - Problem?
- Smart compiler might be able to turn some cases into iterative loop.
- …but don't count on it.
  - VivadoHLS will not

Satisfied?

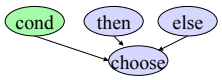
- Q: Satisfied with implementation this is producing?

Beyond Basic Blocks

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

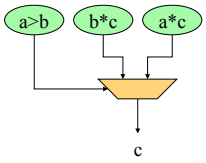
Simple Control Flow

- If (cond) { … } else { …}
- Assignments become conditional
- In simplest cases (no memory ops), can treat as dataflow node



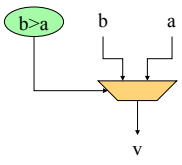
Simple Conditionals

```
if (a>b)
  c=b*c;
else
  c=a*c;
```



Simple Conditionals

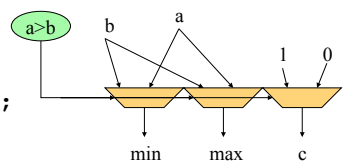
```
v=a;
if (b>a)
  v=b;
```



- If not assigned, value flows from before assignment

Simple Conditionals

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

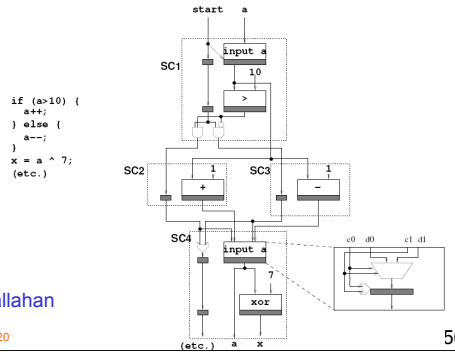


- May (re)define many values on each branch.

Preclass G

- Finish drawing graph for preclass g

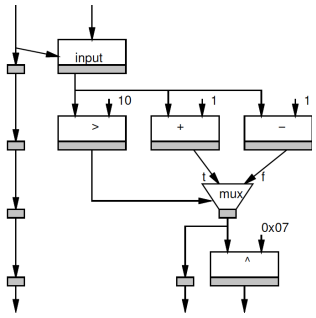
Recall: Basic Blocks for if/then/else



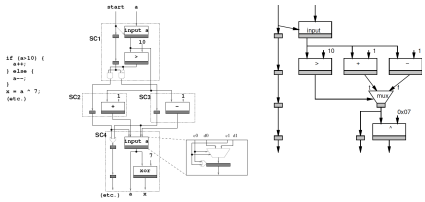
Source: Callahan

Mux Converted

if (a>10)
  a++;
else;
  a—
x=a^0x07

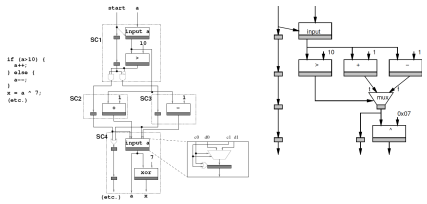


Height Reduction



- 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

Mux Conversion and Memory

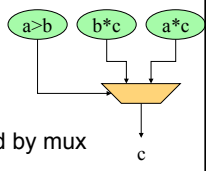
- What might go wrong if we mux-converted the following:
- If (cond)
  - *a=0
- Else
  - *b=0
- Don't want memory operations in non-taken branch to occur.

Mux Conversion and Memory

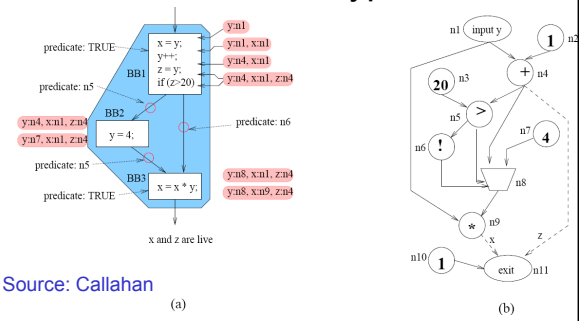
- If (cond)
  - *a=0
- Else
  - *b=0
- Don't want memory operations in non-taken branch to occur.
- **Conclude:** cannot mux-convert blocks with memory operations (without additional care)

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

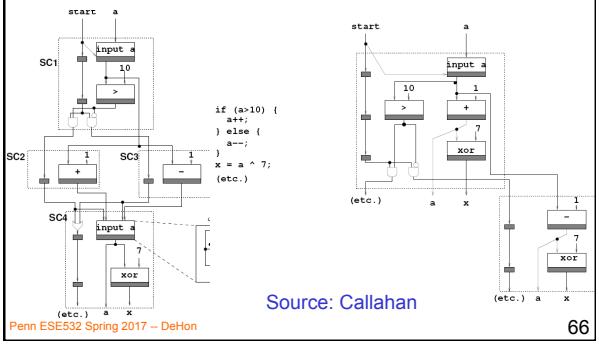


Source: Callahan

Hyperblock Benefits

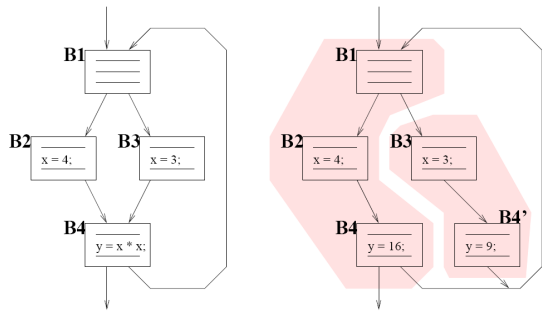
- More code → typically more parallelism
  - Shorter critical path
- Optimization opportunities
  - Reduce work in common flow path
  - Move logic for uncommon case out of path
    - Makes smaller faster

Common Case Height Reduction



Source: Callahan

Common-Case Flow Optimization



Source: Callahan

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);

Additional Concerns?

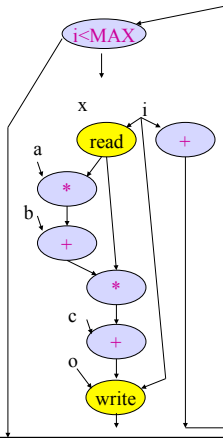
What are we still not satisfied with?

- 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<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;

Unrolling

- If MAX=4:

o[0]=(a*x[0]+b)*x[0]+c;
o[1]=(a*x[1]+b)*x[1]+c;
o[2]=(a*x[2]+b)*x[2]+c;
o[3]=(a*x[3]+b)*x[3]+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;

Unrolling

- If MAX=4:

```
o[0]=(a*x[0]+b)*x[0]+c;
o[1]=(a*x[1]+b)*x[1]+c;
o[2]=(a*x[2]+b)*x[2]+c;
o[3]=(a*x[3]+b)*x[3]+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;
```

Benefits?

Unrolling

- If MAX=4:

```
o[0]=(a*x[0]+b)*x[0]+c;
o[1]=(a*x[1]+b)*x[1]+c;
o[2]=(a*x[2]+b)*x[2]+c;
o[3]=(a*x[3]+b)*x[3]+c;
```

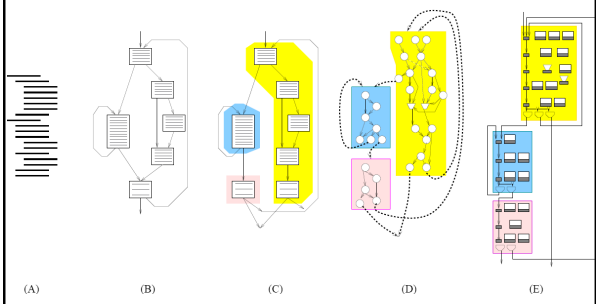
```
for (i=0;i<MAX;i++)
    o[i]=(a*x[i]+b)*x[i]+c;

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.

Unroll

- Vivado HLS has pragmas for unrolling
- UG901: Vivado HLS User's Guide
  - P180—229 for optimization and directives
- **#pragma HLS UNROLL factor=…**

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 registers and gates)

Big Ideas:

- C (or any programming language) specifies a computation
- Can describe spatial computation
  - Has some capabilities that don't make sense in hardware
    - Shared memory pool, malloc, recursion
  - Watch for unintended sequentialization
- C for spatial coded differently from C for processor
  - …but can still run on processor

Admin

- Reading or Monday on Web
- HW5 due Friday