ESE5320: System-on-a-Chip Architecture

Day 9: October 2, 2023

High-Level Synthesis (HLS)

C-to-gates

More accurate: C-for-gates

Today

- Motivation
- Spatial Computations from C specification
  - Variables and expression (pre-lecture)
  - Simple Conditionals (Part 1)
  - Functions (part 2)
  - Globals
  - Loops and Arrays (Part 3)

Message

- C (or any programming language) specifies a computation
- C can describe spatial computation
  - A dataflow graph with physical operators for each operation
- Underlying semantics is sequential
  - Watch for unintended sequentialization
  - Write C for spatial differently than 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 → see in CIS4710, CIS5710
  - Directly in schematics

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?

Three Perspectives

1. How express spatial/hardware computations in C
   – May want to avoid some constructs in C
2. How express computations
   – Hopefully, equally accessible to spatial and sequential implementations
3. Given C code: how could we implement in spatial hardware
   – Some corner cases and technicalities make tricky

[copy to board]

7

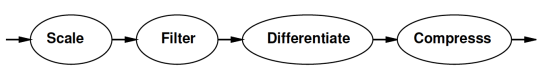
Advantage

- 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
    - …without spending time in place and route
      - …which you soon see is slow…
  - Explore hardware/software tradeoffs by targeting same code to either hardware or software

8

Context

- C most useful for describing behavior of operators



- C alone doesn't naturally capture task parallelism

9

Preclass F

- Ready for preclass f?
- Skip to preclass f

10

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…

11

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…

12

C Primitives
Comparison Operators

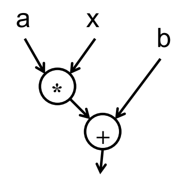
• Less Than	a < b
• Less Than or Equal To	a <= b
• Greater Than	a > b
• Greater Than or Equal To	a >= b
• Not Equal To	a != b
• Equal To	a == b
• Logical Negation	!a
• Logical AND	a && b
• Logical OR	a || b

Things might have a hardware operator for…

13

Expressions:
combine operators

- a*x+b



A connected set of operators
→ Graph of operators

14

Expressions:
combine operators

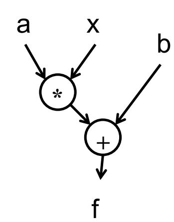
- a*x+b
- a*x*x+b*x+c
- a*(x+b)*x+c
- ((a+10)*b < 100)

A connected set of operators
→ Graph of operators

15

C Assignment

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

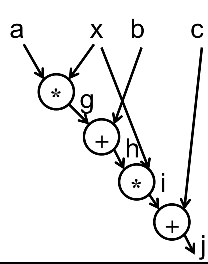


16

Straight-line code

- a sequence of assignments
- What does this mean?

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



17

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;
```

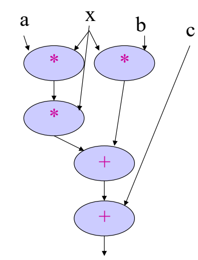

18

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**.

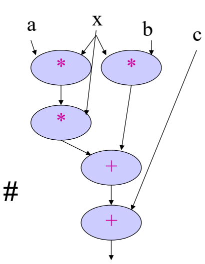
Dataflow

- 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

```
int f(int a, int b)
{
  int t, c, d;

  a=a&(0x0f);
  b=b&(0x0f);
  t=b+3;
  c=a^t;
  t=a-2;
  d=b^t;
  return(d);
}
```


Straight Line Code

- C is fine for expressing straight-line code and variables
  - Has limited data types
    - Address with tricks like masking
    - Address with user-defined types

Optimizations can probably expect compiler to do

- 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);

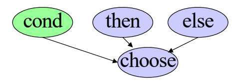
Conditionals

- What can we do for simple conditionals?

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

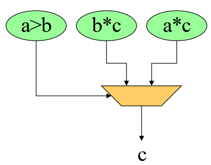

Simple Control Flow

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



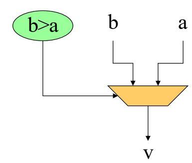
Simple Conditionals: Mux Conversion

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



Simple Conditionals

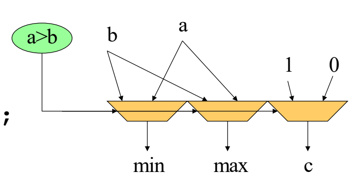
```
v=a;
if (b>a)
  v=b;
```



- 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

- Graph for preclass g as mux-conversion?

```
int g(int a, int b)
{
  int t, c, d;
  // same as above
  a=a&(0x0f);
  b=b&(0x0f);
  t=b+3;
  c=a^t;
  t=a-2;
  d=b^t;
  //added (not in f)
  if (a<b)
     d=c;
  // end added
   return(d);
}
```

Part 2

Functions and Globals

Function Call

- What computation is this describing?

```
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]);
```

- What role does the function call play?

Inline Transformation

- Inline a function
  - Copy the body of function
  - Into the point of call
  - Replacing the function arguments
  - With the arguments supplied in the call

```
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]);
```


Inline

```
int p(int a)
   return(a*a+2*a-1);

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

```
for(i=0;i<MAX;i++)
   D[i]=A[i]*A[i]+2*A[i]-1
       - (B[i]*B[i]+2*B[i]-1);
```

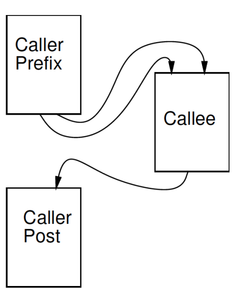
```
for(i=0;i<MAX;i++)
   D[i]=p(A[i]-B[i]);
```

```
for(i=0;i<MAX;i++)
   D[i]=(A[i]-B[i])
       *(A[i]-B[i])
      +2*((A[i]-B[i])-1
```

Functions provide descriptive convenience and compactness.
…but don't need to force implementation.

Treat as data flow

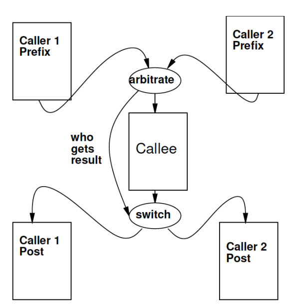
- Implement function as an operation
- Send arguments as input tokens
- Get result back as token



Functions provide potential division between substrates?

Assign different functions to different substrate (proc, fpga)

Shared Function




```
F1(A,B);
 // Transpose(A,Aprime);
 // matmul(Aprime,c1,B);
F2(B,C);
 // matmul(B,c2,Cpre);
 // normalize(Cpre,C);
```

```
if (A<B)
  {
    matmul(A,c1,B);
  }
else
  {
    matmul(D,c3,E);
  }
```

Functions express shared operators.

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?
    - Try Inline?
    - How much hardware do we build?
- Smart compiler might be able to turn some cases into iterative loop.
- …but don't count on it.
  - Vitis HLS will not

Global Variables

- Variables not declared in a function resolve to outer context

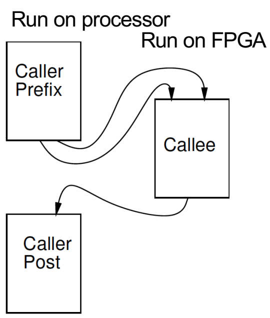
```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }

void f2(int *A) {
   while (A[a]!=0);
      a++;
}

f2(input);
isum=f1(input);
```


Treat as data flow

Functions provide potential division between substrates.

- Impact on global variables?

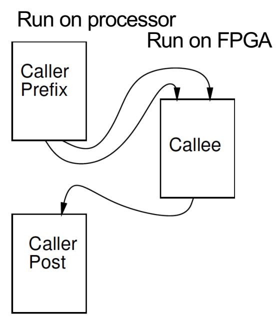
```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }
void f2(int *A) {
   while (A[a]!=0);
      a++; }
f2(input);
isum=f1(input);
```



Treat as data flow

Functions provide potential division between substrates.

- Impact on global variables?
- Correct thing
  - Reflect change in variable between substrates
- Evidence Vitis HLS
  - Not synchronized with host C on globals



Global Variables

- Globals generally considered bad coding practice
  - Obfuscate flow of data even for human
- Avoid Gobals
- With hardware, have extra reason avoid

```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }

void f2(int *A) {
   while (A[a]!=0);
      a++;
}

f2(input);
isum=f1(input);
```


Global Variables

Bad

```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }

void f2(int *A) {
   while (A[a]!=0);
      a++;
}

f2(input);
isum=f1(input);
```

Better

```
int f1(int *A, int len) {
   for (int i=0;i<len;i++)
      sum+=A[i];
   return(sum); }

int f2(int *A) {
   int len=0;
   while (A[len]!=0);
      len++;
   return(len)
}

len=f2(input);
isum=f1(input,len);
```

Part 3

Loops and Arrays

Loops…

- From an *express computation* standpoint, have several roles
  – Compact code
  – Unbounded computation
- From *describe hardware*
  – Compact expression of parallel hardware
  – Express pipelines
  – Express data-level parallelism
  – Express area-time tradeoff

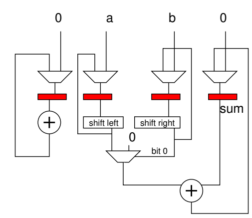
Loop Compact Expression

- What express?
  – Sequential, fully unrolled, partially unrolled?

```
sum=0;
for (i=0;i<32;i++) {
    sum+=(0-(b%2)) & a;
    b=b>>1;
    a=a<<1;
    }
```

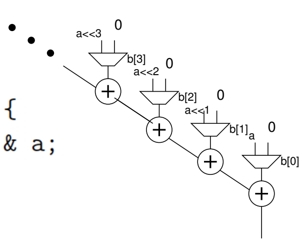

Sequential

```
sum=0;
for (i=0;i<32;i++) {
    sum+=(0-(b%2)) & a;
    b=b>>1;
    a=a<<1;
    }
```



Spatial = fully unrolled

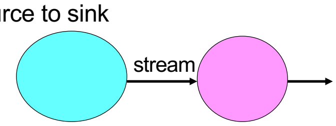
```
sum=0;
for (i=0;i<32;i++) {
    sum+=(0-(b%2)) & a;
    b=b>>1;
    a=a<<1;
    }
```



Day 5

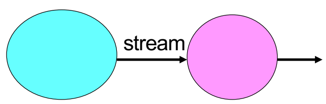
Stream

- Logical abstraction of a persistent point-to-point communication link between operators
  – Has a (single) source and sink
  – Carries data presence / flow control
  – Provides in-order (FIFO) delivery of data from source to sink



Stream

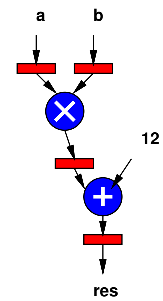
- For the moment assume way to read and write to streams:
  - stream.read() – return next value on stream
  - stream.write(val); put val onto stream



49

Unbounded, Pipelined Operator

What C code describe?

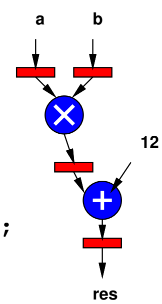


50

Unbounded, Pipelined Operator

What describe?

```
int c=12;
while(true)
  {
    int aval=astream.read();
      int bval=bstream.read();
      int res=aval*bval+c;
      resstream.write(res);
   }
```



51

With function call, loop in function

```
int c=12;
while(true)
   {
     int aval=astream.read();
     int bval=bstream.read();
     int res=multiply(aval,bval)+c;
     resstream.write(res);
    }
```

```
sum=0;
for (i=0;i<32;i++) {
     sum+=(0-(b%2)) & a;
     b=b>>1;
     a=a<<1;
     }
```


52

Compact Expression: Arrays

- Useful to be able to refer to different values (a large number of values) with the same code.
- Arrays + Loops: give us a way to do that
- Useful:
  - general expression
  - hardware description

53

Compact Expression: Arrays+Logic

- Vector sum:
  - c3=a3+b3; c2=a2+b2; c1=a1+b1; c0=a0+b0;
  - for(i=0;i<3;i++) c[i]=a[i]+b[i];
- Chose small length to fit non-array on slide
  - #define K 16
  - for(i=0;i<K;i++) c[i]=a[i]+b[i];

54

Compact Expression: Arrays+Logic

- Dot Product:
  - Y=a3*b3+a2*b2+a1*b1+a0*b0;
  - Y=0; for(i=0;i<3;i++) Y+=a[i]*b[i];

Compact Expression: Arrays+Logic

- Vector sum:
  - c3=a3+b3; c2=a2+b2; c1=a1+b1; c0=a0+b0;
  - for(i=0;i<3;i++) c[i]=a[i]+b[i];
- These array elements may be nodes in dataflow graph, just like the variables we saw for function f
  - Express large dataflow graphs
  - Make area-time choices for implementation

Foreshadowing: C Array Challenge

- C programmers think of arrays as memory (or memory as arrays)
  - …and sometimes we will want to
- Be careful understanding (and expressing) arrays that don't have to be memories
  - …and treated with memory semantics

Loop Interpretations

- What does a loop describe?
  1. Sequential behavior [when to execute]
  2. Spatial construction [when create HW]
  3. Data Parallelism [sameness of compute]
- We will want to use for all 3
- Sometimes need to help the compiler understand which we want

Easy Loop (for contrast)

```
for (i=0;i<10;i++)
    sum+=a[i];
```

- How many times loop execute?
- If unroll, which i for each loop instance?

Loop Bounds

- Loops without constant bounds

```
while (sum+a[i]<100) {
  sum+=a[i];
  i++; }
```

- How many times loop execute?
- Typically forces sequentialization
  - Cannot unroll into hardware

Loop Increment

- Loops with variable increment also force sequentialization

```
for (i=0;i<100;i+=f[i])
  { b[i]=a[i]; sum+=a[i]; }
```

- What are values of i for which evaluate body?

Penn ESE5320 Fall 2023-- DeHon 61

61

Loop Interpretations

- What does a loop describe?
  - Sequential behavior [when execute]
  - Spatial construction [when create HW]
  - Data Parallelism [sameness of compute]
- We will want to use for all 3
- C allows expressive loops
  - Some expressiveness
    - Not compatible with spatial hardware construction

Penn ESE5320 Fall 2023-- DeHon 62

62

Unroll

- Vitis HLS has pragmas for unrolling
- UG1399: Vitis HLS User's Guide
- **#pragma HLS UNROLL factor=…**
- Use to control area-time points
  - Use of loop for spatial vs. temporal description
- In general – *pragmas* – directives to the compiler telling how to compile; does not change meaning of program

Penn ESE5320 Fall 2023-- DeHon 63

63

Big Ideas:

- C (any prog lang) specifies a computation
- Can describe spatial computation
  - Has some capabilities that don't make sense in hardware
    - Shared memory pool, globals, recursion
  - Watch for unintended sequentialization
- C for spatial is coded differently from C for processor
  - …but can still run on processor
- Good for leaf functions (operations)
  - Limiting for full task

Penn ESE5320 Fall 2023-- DeHon 64

64

Admin

- Feedback, incl. HW4
- Reading for Wednesday online
- HW5 due Friday
  - Several long compiles; start early

Penn ESE5320 Fall 2023-- DeHon 65

65