ESE532: System-on-a-Chip Architecture

Day 9: September 30, 2019
High Level Synthesis (HLS)
C-to-gates
Maybe: C-for-gates

Today

- Motivation
- Spatial Computations from C specification
  - Variables and expression (skip?)
  - Simple Conditionals
  - Loops
  - Functions
  - Arrays
  - Memory
- Complexities from C semantics

Message

- C (or any programming language) specifies a computation
- 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 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 → see in CIS371, CIS501
  - 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?

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]

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 are beginning to see now…
  - Explore hardware/software tradeoffs by targeting same code to either hardware or software

Context

- C most useful for describing behavior of leaf operators

- C alone doesn't naturally capture task parallelism

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…

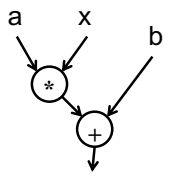
C Primitives
Comparison Operators

- 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

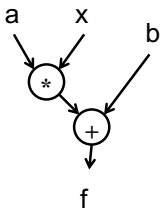
Expressions: combine operators

- 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

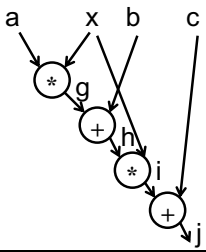
- Basic assignment statement is:
  Location = expression
- f=a*x+b



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

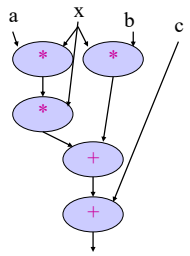
```
t=a*x;   t=a*x;
r=t*x;   r=t*x;
t=b*x;           t=b*x;
r=r+t;     r=r+t;
r=r+c;     r=r+c;
```

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

Dataflow

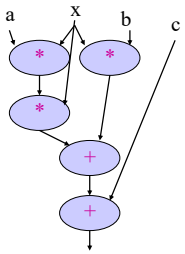
- Can turn sequential assignments into dataflow graph through def→use connections

```
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

```
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);

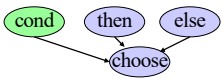
Conditionals

- What can we do for simple conditionals?

```
if (a<b)
   res=b-a
Else
   res=a-b
```

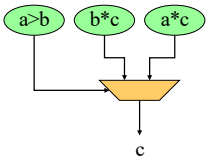

Simple Control Flow

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



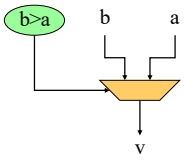
Simple Conditionals

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



Simple Conditionals

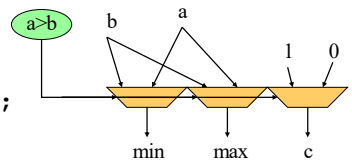
```
v=a;
if (b>a)
  v=b;
```



- 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

- 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);
}
```

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]);
```

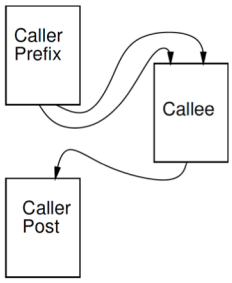
```
for(i=0;i<MAX;i++)
    D[i]=A[i]*A[i]+2*A[i]-1
       - (B[i]*B[i]+2*B[i]-1);
```

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

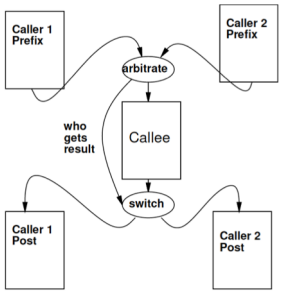
Treat as data flow

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

Functions provide potential division between substrates?



Shared Function



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

Global Variables

- Variables not declared in a function resolve to outer context

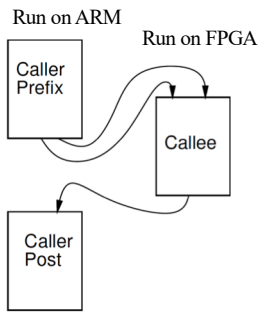
```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }

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?

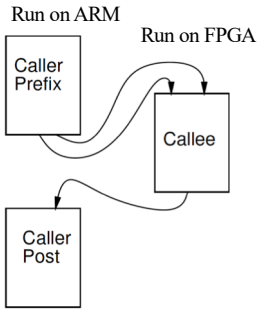
```
int a=0;
int f1(int *A) {
   for (int i=0;i<a;i++)
      sum+=A[i];
   return(sum); }
void f2(int *A) {
   while (A[a]!=0);
      a++; }
f2(input);
isum=f1(input);
```



Treat as data flow

Functions provide potential division between substrates.

- Impact on global variables?
- Correct thing
  - Reflect change in variable between substrates
- Evidence Vivado HLS
  - Does not do that



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) {
   while (A[a]!=0);
      len++;
   return(len)
}

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


Loops…

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

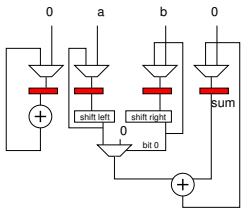
Loop Compact Expression

- 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;
    }
```

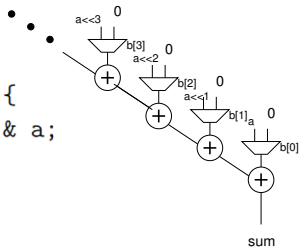

Sequential

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



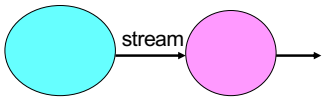
Spatial = fully unrolled

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



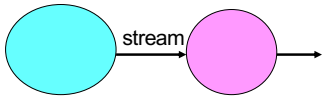
Stream

- 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



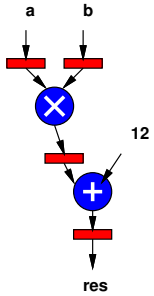
Stream

- 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



Unbounded, Pipelined Operator

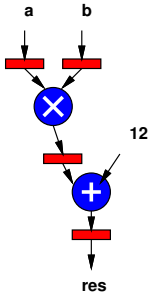
What C code describe?



Unbounded, Pipelined Operator

What describe?

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



With function call, loop in function

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

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


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

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];

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

Loop Bounds

- Loops without constant bounds

```
while (sum+a[i]<100) {
  sum+=a[i];
    b[i]=a[i]>>2;
    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?

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

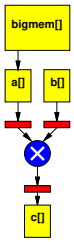
Unroll

- Vivado HLS has pragmas for unrolling
- UG901: Vivado HLS User's Guide
  - P180—229 for optimization and directives
- **#pragma HLS UNROLL factor=…**
- Use to control area-time points
  - Use of loop for spatial vs. temporal description

Arrays as Memory Banks

- Hardware expression: Sometimes we will want to describe computations with separate memory banks

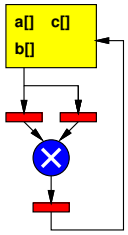
```
int a[1024], b[1024],
    c[1024];
for(i=0;i<1024;i++)
  a[i]=bigmem[offset+i];
for (i=0;i<1024;i++)
  c[i]=a[i]*b[i];
```



Arrays as Memory Banks

- If single memory has only one port
  - Can perform only one memory operation per cycle
  - What II if a, b, c all in bigmem?

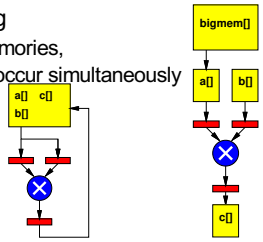
```
for (i=0;i<1024;i++)
  c[i]=a[i]*b[i];
```



Physical Memory Port as Limited Shared Resource

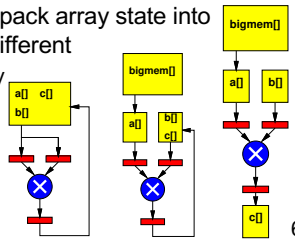
- Typically single memory port
  - Must sequentialize on use of memory port
  - Reason for banking
    - Put in separate memories, so operations can occur simultaneously

Ultra96 DRAM 1 port
Virtex BRAM 2 ports



Arrays as things to put in Memory Banks

- Computational expression:
  - sometimes it is useful to express computation
  - **then** decide how to pack array state into memory banks for different
    - Hardware availability
    - Area-Time tradeoffs



Arrays as Inputs and Outputs

- Computational Expression: arrays are often a natural way of expression set of inputs and outputs

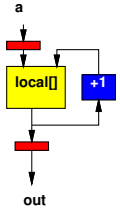
```
int c=12;
while(true)
   {
     int aval=astream.read();
     int bval=bstream.read();
     int res=a*b+c;
     resstream.write(res);
     }
```

```
void op(int a[BLOCK], int
  b[BLOCK], int out[BLOCK]) {
    for (i=0;i<BLOCK;i++)
     {
       out[i]=a[i]*b[i]+c;
     }
}
```


Arrays as Local Memory

- Hardware/Computational expression: natural way of describing local state

```
hist(int a[SIZE], out[EVENTS]) {
    int local[EVENTS];
    for(i=0;i<EVENTS;i++)
      local[i]=0;
    for(i=0;i<SIZE;i++)
      local[a[i]]++;
    for(i=0;i<EVENTS;i++)
      out[i]=local[i];
}
```

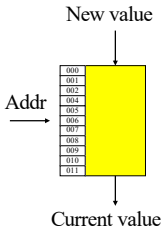


Array Interpretations

- What does an array describe?
  1. Compact expression [write less code]
  2. Memory banks [where place data]
  3. Things put in separate memory banks
  4. Local memory [not need to be shared]
  5. I/O [source and sink of data]
- We will want to use for all 5
- C allows expressive use of arrays/memories
  - Some expressiveness will inhibit efficient hardware

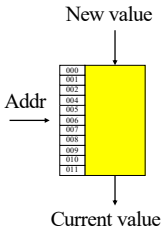
C Memory Model

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



Challenge: C Memory Model

- One big linear address space of locations
- Assumes all arrays live in same memory
- Assumes arrays may overlap?

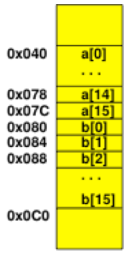


Example

- Assume a, b live in same memory
- Placed in sequence as shown
- What happens when

```
int a[16];
int b[16];
```

  - Read from a[17]
  - Read from b[-2]
- Can inhibit separation into memory banks, parallelism



Memory Operation Challenge

- Memory is just a set of location
- But **memory expressions** in C can refer to variable locations
  - Does A[i], B[j] refer to same location?
  - A[f(i)], B[g(j)] ?
- Can inhibit banking, parallelism
  - Or add expensive interconnect

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

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
  - Writes which may redefine the same location cannot be reordered
- True for read/write to single array even if know arrays isolated
  - Does A[B[i]] refer to same location as A[C[i]]?
  - So expression issue broader than C

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

More on Wednesday

- If time permits on Wednesday, more on Sequentialization and Dependencies

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

Admin

- Reading for Wednesday on Web
  - Xilinx HLS documents
- HW5 due Friday
  - Remember several long compiles
  - Get started early