ESE5320:
System-on-a-Chip Architecture

Day 11: October 10, 2022
Coding HLS for Accelerators

Midterm average: 60

Midterm

- Solutions posted
- Harder midterm
  - Critical Path problem for whole loop was definitely too much
- Mostly diagnostic and warmup
- Will take max(midterm,final) for midterm grade

Previously

- We can describe computational operations in C
  - Primitive operations (add, sub, multiply, and, or)
  - Dataflow graphs primitives
  - To bit level
  - Conditionals and loops
  - Function abstraction
  - Loops, Arrays

Today

- Arrays and Memory Sequentialization – Part 1
- Controlling Parallelism in Vitis HLS C – Part 2
- Controlling Memories in Vitis HLS C – Part 3
- Time permitting – Part 4
  - malloc, pointers,
- Supplement – Part 5
  - more dependencies

Message

- Can specify HW computation in C
- Vitis HLS gives control over how design mapped (area-time, streaming…)
- Code may need some care and stylization to feed data efficiently
- Read UG 1393 – Profiling, Optimizing, and Debugging > Optimizing C++ Kernels

Day 9

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

Arrays and Memories

7

Loop Interpretations

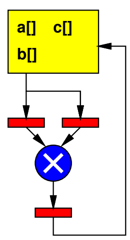
- What does a loop describe?
  - Sequential behavior [when execute]
  - Spatial construction [when create HW]
  - Data Parallelism [sameness of compute]
- We will want to use for all 3
- C allows expressive loops
  - Some expressiveness
    - Not compatible with spatial hardware construction

8

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];
```

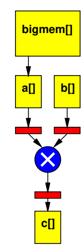


9

Arrays as Memory Banks

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

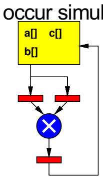
```
int a[1024], b[1024],
     c[1024];
for(i=0;i<1024;i++)
   a[i]=bigmem[offset+i];
for (i=0;i<1024;i++)
   c[i]=a[i]*b[i];
```

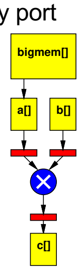


10

Physical Memory Port as Limited Shared Resource

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



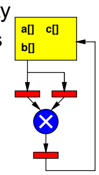


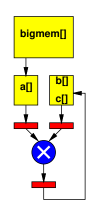
Ultra96 DRAM 1 port
Virtex BRAM 2 ports

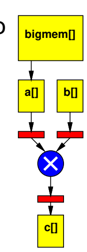
11

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





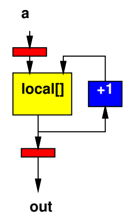


12

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];
}
```



13

Arrays as Inputs and Outputs

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

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


14

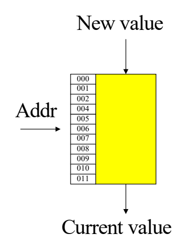
Array Interpretations

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

15

C Memory Model

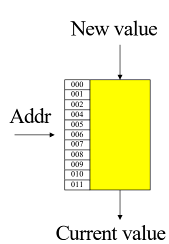
- One big linear address space of locations
- Most recent definition to location is value
- Sequential flow of statements



16

Challenge: C Memory Model

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



17

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

18

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
- Challenge for single array
  - Does A[B[i]] refer to same location as A[C[i]]?
  - So expression issue broader than C

C Memory/Pointer Sequentialization

- Must preserve ordering of memory operations
- Challenge for single array:
  - Does A[B[i]] refer to same location as A[C[i]]?
- Challenge multiple arrays:
  - Out-of-bounds reference for one array reference another?
  - void fun(int a[100], int b[100], int c[100])
    - { for (int i=0;i<100;i++) c[i]=a[b[i]]; }
  - What if called: fun(a,perm,a);

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 at end of lecture

- More on Sequentialization and Dependencies
  - Slides there to review
  - Won't cover

Vivado HLS Mapping Control: Compute Parallelism Loops, Dataflow

Part 2

Assume f?(in,out)

Dataflow Graph

- What dataflow graph does this describe?

```
while(true) {
        i=read_input();
        fA(i,t1);
        fB(t1,t2);
        fC(t2,out);
        write_output(out);
}
```

Define: Pragma

- Pragma – directive to compiler
  - Usually compiler specific
    - So ignored by compilers that don't support (where not relevant)
    - Gives compiler hints/direction on how to compile
    - Should not change meaning of computation
      - Should compute the same thing
      - …but may compute differently
- Use Pragmas to direct Vitis HLS on how to compile our code for FPGA

Vivado HLS Pragma DATAFLOW

- Enables streaming data between functions and loops
  - Don't have to wait for entire array to be produced to move data and start downstream computation
- Allows concurrent streaming execution
- Requires data be produced/consumed sequentially
  - i.e. can connect with FIFO; not need reorder

Dataflow with Arrays

```
int i[100];
int t1[100],t2[100];
int out[100];
while(true) {
      read_input(i,100);
       fA(i,t1);
       fB(t1,t2);
       fC(t2,out);
       write_output(out,100);
}
```


Streamable

- When process input and output in order

```
void fA (int in[100], int out[100])
{
   out[0]=in[0];
   for (int i=1;i<100;i++)
     out[i]=(in[i]+in[i-1])/2;

}
```


Cannot Stream Input

- Why?

```
void fB (int in[100], int out[100])
   {
      for (int=0; i<100;i++)
         out[i]=in[100-i];
    }
```


Streamable?

- Can stream input?
- Can stream output?

```
void fC (int in[100], int out[100])
   {
      for (int=0; i<100;i++)
         out[i]=0;
      for (int=0; i<100;i++)
         out[in[i]%100]++;
   }
```

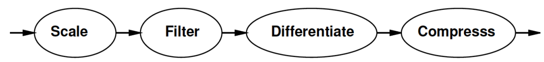
Vivado HLS Pragma DATAFLOW

- Enables streaming data between functions and loops
- Allows concurrent streaming execution
- Requires data be produced/consumed sequentially
  - i.e. can connect with FIFO; not need reorder
- Useful to use stream data type between functions – communicates sequence
  - hls::stream<TYPE>

Streaming Operations

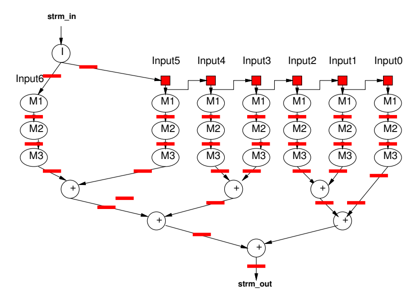
- Functions can have stream inputs and outputs
  - Must pass as pointers

    hls::stream<Type> &strm
- Vivado HLS expressiveness to define hardware streaming operation pipelines




```
void stream_filter (
            hls::stream<uint16_t> &strm_out,
            hls::stream<uint16_t> &strm_in
          )
  while(true) {
  yout=0;
  Input5=Input6;
  Input4=Input5;
  Input3=Input4;
  Input2=Input3;
  Input1=Input2;
  Input0=Input1;
  strm_in.read(Input0);
  Sum = Coefficients_0 * Input0 +
        Coefficients_1 * Input1 +
        Coefficients_2 * Input2 +
        Coefficients_3 * Input3 +
        Coefficients_4 * Input4 +
        Coefficients_5 * Input5 +
        Coefficients_6 * Input6;
  strm_out.write(Sum>>8);
  }
```


stream_filter Pipeline



Dataflow Streaming

- Works between loops, as well

Between Loops

```
int data_in[N],data_out[N*256];
hls::stream<int> ystream;
short val,res,copies;
int current;

#pragma HLS dataflow

for (i=0;i<N;i++) {
   pair=data_in[i];
   copies=(pair>>16)&0x0ff;
   val=pair&0x0ffff;
   for (j=0;j<copies;j++)
      ystream.write(val);
   }

for (int i=0;i<N*256;i++)
  {
     ystream.read(res);
     current=current+res;
     data_out[i]=current;
  }
```

Vivado HLS Pragma PIPELINE

- Direct a function or loop to be pipelined
- Ideally start one loop or function body per cycle
  - Can control II

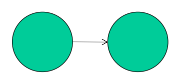

```
for (i=0;i<N;i++)
   yout=0;
   for (j=0;j<K;j++)
      #pragma HLS PIPELINE
         yout+=in[i+j]*w[j];
   y[i]=yout;
```

Which solution from preclass 3?

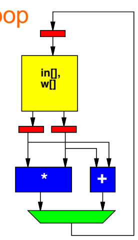
Dataflow and pipelining

- Dataflow allows coarse-grained pipelining among loops and functions
- Pipeline causes loop bodies to be pipelined

Dataflow and Pipelining

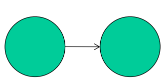


- Cycles for top loop unpipelined?

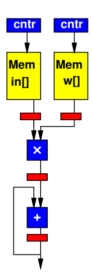



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

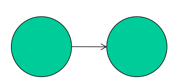


- Cycles for top loop pipelined?

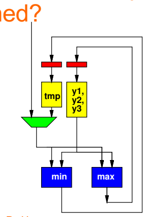



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

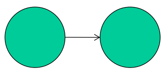


- Cycles for bottom loop unpipelined?

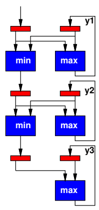



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```

Dataflow and Pipelining



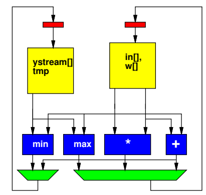
- Cycles for bottom loop pipelined?




```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

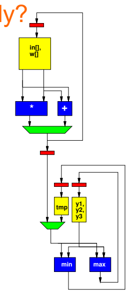
- Composite time, no dataflow, no pipelining?




```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

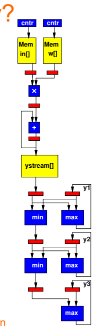
- Composite time dataflow only?



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

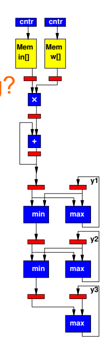
- Composite time pipelining only?



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Dataflow and Pipelining

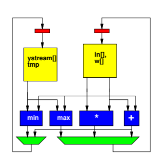
- Composite time dataflow and pipelining?

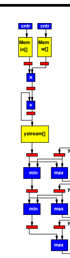


```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```

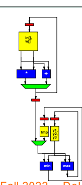

Compare Cases

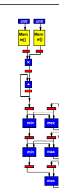
No Dataflow





Dataflow

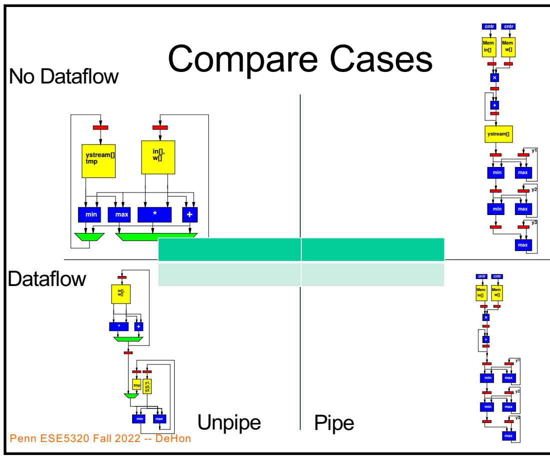




Unpipe

Pipe

Compare Cases



Unroll

- Vivado HLS has pragmas for unrolling
- UG902: Vivado Design Suite HLS User's Guide
  - P139—142 (2020.1, 2018.3)
- **#pragma HLS UNROLL factor=…**

  https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf
  https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis
- Use to control area-time points
  - Use of loop for spatial vs. temporal description

Vivado HLS Pragma UNROLL

- Unroll loop into spatial hardware
  - Can control level of unrolling
- Any loops inside a pipelined loop gets unrolled by the PIPELINE directive


```
for (i=0;i<N;i++)
   yout=0;
   for (j=0;j<K;j++)
      #pragma HLS UNROLL
      yout+=in[i+j]*w[j];
   y[i]=yout;
```

Which solution from preclass 3?

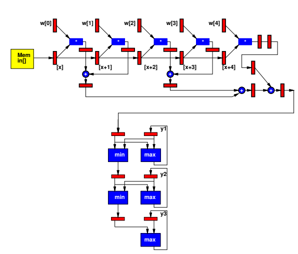
With Pipelining

```
for (i=0;i<N;i++)
   yout=0;
   #pragma HLS PIPELINE
   for (j=0;j<K;j++)
      yout+=in[i+j]*w[j];
   y[i]=yout;
```

Which solution from preclass 3?

Dataflow, Unrolling, & Pipelining

- Cycles unroll K-loop, dataflow, pipeline?



```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y1);
  tmp=min(d,y1);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```

Vivado HLS Pragma INLINE

- Collapse function body into caller
  - Eliminates interface code
  - Allows optimization of inline code
- Recursive option to inline a hierarchy
  - Maybe useful when explore granularity of accelerator

Vivado HLS Mapping Control: Memories

Part 3

Zynq BRAM

- 36Kb of memory
  - Configurable width up to 72b
    - 512x72 or … 32Kx1
  - Dual port
- Can be operated as 2x18Kb memory banks
  - Configurable width up to 36b
    - 512x36 or … 16Kx1
  - Each memory dual port
- Xilinx UG573, UltraScale Architecture Memory Resources User Guide

Vivado HLS Pragma ARRAY_PARTITION

- Spread out array over multiple BRAMs
  - By default placed in single BRAM
    - At most 2 ports
  - Use to remove memory bottleneck that prevents pipelining (limits II)

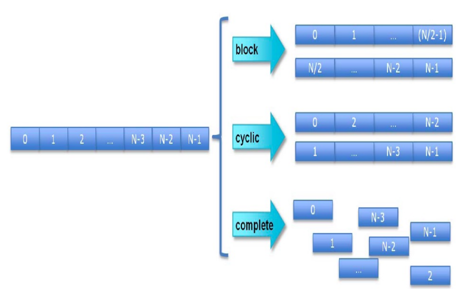
Memory Bottleneck Example

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

What problem if put mem in single BRAM?

Xilinx UG1197 (2017.1) p. 50

Array Partition



Xilinx UG902 p. 195 (145 in 2017.1 version)

Array Partition Example

#pragma ARRAY_PARTITION variable=mem cylic factor=4

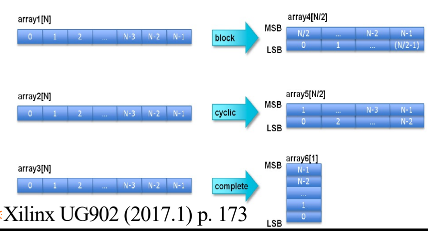
```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

 Xilinx UG902 p. 91

Vivado HLS Pragma ARRAY_RESHAPE

- Pack data into BRAM to improve access (reduce BRAMs)
  - May provide similar benefit to partitioning without using more BRAMs


```
void foo (...) {
int  array1[N];
int  array2[N];
int  array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```



 Xilinx UG902 (2017.1) p. 173

BRAM can be configured for 72b wide output

How fix if dint_t is 16b?

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

 Xilinx UG902 p. 91

Array Reshape Example

#pragma ARRAY_RESHAPE variable=mem cylic factor=4 dim=1
(if din_t 16b)

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

 Xilinx UG902 p. 91

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

HLS Pragma Summary

- pragmas allow us to control hardware mapping
  - How interpret loops (spatial hw vs. temporal)
  - How arrays get mapped to memories
  - How treat function calls
  - Turn area-time knobs
- Could have rewritten code by hand
  - Unroll, separate arrays…
  - Pragmas automate; we just need to provide instruction

Memory Allocation Part 4

Simple answer: "Don't do it!"

Skip to Wrapup

Demand for malloc()

- Data-dependent object (array) size
- Data-dependent number of objects

Hardware Memory

- Typically small, fixed, local memory blocks
  - E.g. 36Kb BRAMs
- Reuse memory blocks
  - Not allocate new blocks
  - Cannot make data-dependent memory sized blocks
  - Cannot hold arbitrary-sized data

No malloc()

- Generally don't want to use malloc with
  - Hardware Accelerated functions
  - Real-time computations
- Vivado HLS won't let you use malloc()
  - For C running on FPGA array
- **Instead:** statically declare arrays of maximum size data may be

Pointer Passing

Be careful…

Pointer Passing

- What does it mean to pass a pointer into a function?

73

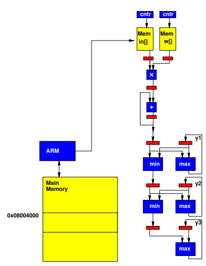
Pointer Passing Interpretations

- Multiple uses we may want to express
  1. Specify which data to work on
     - Ok to copy that data to private accelerator memory and work with it
     - But, how much data to copy? (length)
  2. Want to mutate data and have other (parallel) tasks see it
     OR want to see data mutated by other (parallel tasks)
     - Not OK to copy to private accelerator memory
     - Force use from large, shared memory
     - Forces sequentialization

74

Pointer Passing

- What if accelerator doesn't have access to the memory holding the data pointed to by the pointer?

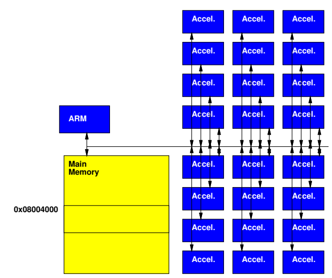


75

Pointer Passing

Maybe only reading data that will not change.

What happens if we give accelerators access to common memory holding data for pointer, but
- There's only one port into memory
- Memory is 10 cycles away
- And there are 100 accelerators that may need access
- Memory can only handle one memory op per cycle



76

Avoid Pointer Passing

- Tend to copy data into / move data among hardware accelerator memories rather than passing pointers.

77

Memory Sequentialization and Data Dependencies Part 5

(unlikely to cover in class;
Review on own)

Skip to wrapup

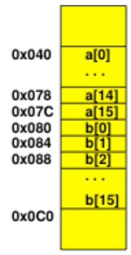
78

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 Consequence

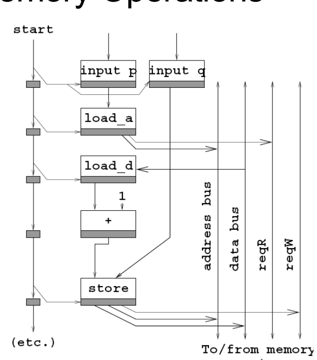
- **Expressions and operations** through variables (whose address is never taken) can be executed at any time
  – Just preserve the dataflow
- **Memory assignments** must execute in strict order
  – Ideally: partial order
  – Conservatively: strict sequential order of C

Forcing Sequencing

- Demands we introduce some discipline for deciding when operations occur
  – Could be a FSM
  – Could be an explicit dataflow token
  – Callahan (reading) 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 (separate banks)?
- Is the compiler smart enough to exploit?

Mux Conversion and Memory

- What might go wrong if we mux-converted the following:

```
if (cond)
   a[i]=0;
else
   b[i]=0;
```


Mux Conversion and Memory

- What might go wrong if we mux-converted the following:

```
if (cond)
   a[i]=0;
else
   b[i]=0;
```

- Don't want memory operations in non-taken branch to occur.

Mux Conversion and Memory

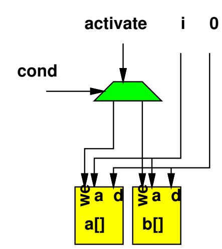
```
if (cond)
   a[i]=0;
else
   b[i]=0;
```

Don't want memory operations in non-taken branch to occur.

- **Conclude:** cannot mux-convert blocks with memory operations (without additional care)

Conditions and Memory

```
if (cond)
   a[i]=0;
else
   b[i]=0;
```



Dependence in Loops

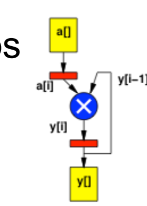
```
for(i=0;i<K;i++)
   Y[i]=a[i]*Y[i-1];
```

If a value needed by one instance of the loop is written by another instance, can create cyclic dependence.

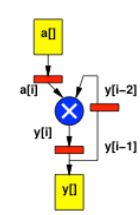
→ limit parallelism (pipeline II)

Dependence in Loops

```
for(i=0;i<K;i++)
   Y[i]=a[i]*Y[i-1];
```




```
for(i=0;i<K;i++)
   Y[i]=a[i]*Y[i-2];
```



Dependence distance same as # registers in cycle.

Dependence Fixed/Predictable?

```
for(i=0;i<K;i++)
   Y[i]=a[i]*Y[i-1]+Y[i-2];

for(i=0;i<K;i++)
   Y[i]=a[i]*Y[b[i]];
```

If dependence data-dependent, forced to sequentialize.

Dependence Fixed/Predictable?

```
for(i=0;i<K;i++)
   Y[i]=a[i]*Y[i-1]+Y[i-2];

for(i=0;i<K;i++)
   Y[i]=a[i]*Y[2*i+3];
```

If dependence linear, aggressive compliers may be able to resolve.

Dependence Fixed/Predictable?

```
for(i=0;i<K;i++)
   Y[i]=
  a[i]*Y[ceil(sqrt(i)*sin(2i))];
```

If dependence too complicated, compiler not solve and will force sequential execution.

Big Ideas

- Can specify HW computation in C
- Create streaming operations
  - Run on processor or FPGA
- Vivado HLS gives control over how map to hardware
  - Area-time point

Admin

- Feedback
- Reading for Wednesday
  - on web and Zynq book
- HW5 due Friday
  - Start early; require slow builds