ECE 636
Reconfigurable Computing
Lecture 12

High-Level Compilation
Overview

• High-level language to FPGA an important research area

• Many challenges

• Commercial and academic projects
  - Celoxica
  - DeepC
  - Stream-C

• Efficiency still an issue. Most designers prefer to get better performance and reduced cost
  - Includes incremental compile and hardware/software codesign
Issues

° Languages
  • Standard FPGA tools operate on Verilog/VHDL
  • Programmers want to write in C

° Compilation Time
  • Traditional FPGA synthesis often takes hours/days
  • Need compilation time closer to compiling for conventional computers

° Programmable-Reconfigurable Processors
  • Compiler needs to divide computation between programmable and reconfigurable resources

° Non-uniform target architecture
  • Much more variance between reconfigurable architectures than current programmable ones

Acknowledgment: Carter
Why Compiling C is Hard

- General Language
- Not Designed For Describing Hardware
- Features that Make Analysis Hard
  - Pointers
  - Subroutines
  - Linear code
- C has no direct concept of time
- C (and most procedural languages) are inherently sequential
  - Most people think sequentially.
  - Opportunities primarily lie in parallel data

Lecture 12: High-Level Compilation

October 17, 2013
Notable FPGA High-Level Compilation Platforms

° Celoxica – Handel-C
  • Commercial product targeted at FPGA community
  • Requires designer to isolate parallelism
  • Straightforward vision of scheduling

° DeepC
  • Completely automated – no special actions by designer
  • Ideal for data parallel operation
  • Fits well with scalable FPGA model

° Stream-C
  • Computation model assumes communicating processes
  • Stream based communication
  • Designer isolates streams for high bandwidth
Handel-C adds constructs to ANSI-C to enable hardware implementation

- synthesizable HW programming language based on ANSI-C
- Implements C algorithm direct to optimized FPGA or outputs RTL from C
Fundamentals

- Language extensions for hardware implementation as part of a system level design methodology
  - Software libraries needed for verification
- Extensions enable optimization of timing and area performance
- Systems described in ANSI-C can be implemented in software and hardware using language extensions defined in Handel-C to describe hardware.
- Extensions focused towards areas of parallelism and communication

Courtesy: Celoxica
Variables

- Handel-C has one basic type - integer
- May be signed or unsigned
- Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers.

```c
void main(void)
{
    unsigned 6 a;
    a=45;
}
```

\[ a = \begin{array}{cccccc}
1 & 0 & 1 & 1 & 0 & 1
\end{array} = 0x2d \]
Timing model

- Assignments and delay statements take 1 clock cycle
- Combinatorial Expressions computed between clock edges
  - Most complex expression determines clock period
  - Example: takes 1+n cycles (n is number of iterations)

```plaintext
index = 0;                      // 1 Cycle
while (index < length){
    if(table[index] = key)
        found=index;     // 1 Cycle
    else
        index = index+1; // 1 Cycle
}
```
Parallelism

° Handel-C blocks are by default sequential
° `par{...}` executes statements in parallel
° `par` block completes when all statements complete
  * Time for block is time for longest statement
  * Can nest sequential blocks in `par` blocks
° Parallel version takes 1 clock cycle
  * Allows trade-off between hardware size and performance

```
Parallel Block

// 1 Clock Cycle
par{
    a=1;
    b=2;
    c=3;
}
```

```
Parallel code

par(i=0;i<10;i++)
{
    array[i]=0;
}
```
Channels

° Allow communication and synchronisation between two parallel branches
  • Semantics based on CSP (used by NASA and US Naval Research Laboratory)
  • unbuffered (synchronous) send and receive

° Declaration
  • Specifies data type to be communicated

```
Chan unsigned 6 c;

{ …
    c!a+1;  //write a+1 to c
    ...    
} { …
    c?b;    //read c to b
    ...    
} 
```
Signals

- A signal behaves like a wire - takes the value assigned to it but only for that clock cycle.
  - The value can be read back during the same clock cycle.
  - The signal can also be given a default value.

```c
// Breaking up complex expressions
int 15 a, b;
signal <int> sig1;
static signal <int> sig2=0; //default value of 0
a = 7;
par
{
    sig1 = (a+34)*17;
sig2 = (a<<2)+2;
b = sig1 + sig2;
}
```
Sharing Hardware for Expressions

- Functions provide a means of sharing hardware for expressions

- By default, compiler generates separate hardware for each expression
  - Hardware is idle when control flow is elsewhere in the program

```c
{...
  x = x*a + b;
  y = y*c + d
}
```

- Hardware function body is shared among call sites

```c
int mult_add(int z, c1, c2) {
  return z*c1 + c2;
}
```

```c
{...
  x = mult_add(x, a, b);
  y = mult_add(y, c, d);
}
```
DeepC Compiler

- Consider loop based computation to be memory limited
- Computation partitioned across small memories to form tiles
- Inter-tile communication is scheduled
- RTL synthesis performed on resulting computation and communication hardware
DeepC Compiler

- Parallelizes compilation across multiple tiles
- Orchestrates communication between tiles
- Some dynamic (data dependent) routing possible.

\[
\begin{align*}
&\text{(b)} & A[i] & B[i] \\
&\Rightarrow & \text{for}(i=0; i<1000; i++) \\
\end{align*}
\]

\[
\begin{align*}
&\text{(c)} & A_3[i] & B_3[i] & A_1[i] & B_1[i] \\
&\Rightarrow & \text{for}(i=0; i<1000; i++) \\
& & A_3[i] = A_3[i] \times B_3[i+1] \\
& & A_1[i+1] = A_1[i+1] \times B_1[i+2] \\
& & A_1[i+2] = A_1[i+2] \times B_1[i+3] \\
& & A_1[i+3] = A_1[i+3] \times B_1[i+4] \\
& & \text{for}(i'=0; i'<1000; i'++) \\
& & A_3[i'] = A_3[i'] \times B_3[i'] \\
& & A_1[i'+1] = A_1[i'+1] \times B_1[i'+1] \\
& & A_1[i'+2] = A_1[i'+2] \times B_1[i'+2] \\
& & A_1[i'+3] = A_1[i'+3] \times B_1[i'+3] \\
& & i' = i' + 1 \\
\end{align*}
\]
Control FSM

- Result for each tile is a datapath, state machine, and memory block
DeepC Results

- Hard-wired case is point-to-point
- Virtual-wire case is a mesh
- RAW uses MIPs processors
Bitwidth Analysis

° Higher Language Abstraction
  • Reconfigurable fabrics benefit from specialization
  • One opportunity is bitwidth optimization

° During C to FPGA conversion consider operand widths
  • Requires checking data dependencies
  • Must take worst case into account
  • Opportunity for significant gains for Booleans and loop indices

° Focus here is on specialization

Courtesy: Stephenson
Arithmetic Operations

Example

```c
int a;
unsigned b;
a = random();
b = random();
```

\[ a: 32 \text{ bits} \quad b: 32 \text{ bits} \]

```
a = a / 2;
```

\[ a: 31 \text{ bits} \quad b: 32 \text{ bits} \]

```
b = b >> 4;
```

\[ a: 31 \text{ bits} \quad b: 28 \text{ bits} \]
Bitmask Operations

° Example

```c
int a;
a: 32 bits
a = random() & 0xff;
a: 8 bits
```
Loop Induction Variable Bounding

- Applicable to for loop induction variables.
- Example

```c
int i;

for (i = 0; i < 6; i++) {
    i: 32 bits
    ...
}
i: 3 bits

i: 3 bits
```
Clamping Optimization

- Multimedia codes often simulate saturating instructions.
- Example
  ```c
  int valpred
  
  valpred: 32 bits
  
  if (valpred > 32767)
      valpred = 32767
  
  else if (valpred < -32768)
      valpred = -32768
  
  valpred: 16 bits
  ```
Solving the Linear Sequence

\[ a = 0 \quad <0,0> \]
for i = 1 to 10
\[ a = a + 1 \quad <1,460> \]
for j = 1 to 10
\[ a = a + 2 \quad <3,480> \]
for k = 1 to 10
\[ a = a + 3 \quad <24,510> \]
....= a + 4 \quad <510,510>

◦ Can easily find conservative range of \(<0,510>\)

◦ Sum all the contributions together, and take the data-range union with the initial value.
FPGA Area

Benchmark (main datapath width)

Area (CLB count)

- Without bitwise
- With bitwise

Lecture 12: High-Level Compilation

October 17, 2013
FPGA Clock Speed
(50 MHz Target)

Without bitwise
With bitwise

XC4000-09 Clock Speed (MHZ)

0 25 50 75 100 125 150

adpcm bubblesort convolve histogram intfir intmatmul jacobi life median mpegcorr newlife parity pmatch sor

Lecture 12: High-Level Compilation

October 17, 2013
Streams-C

- Stream based extension to C
  - Augment C to facilitate stream-based data transfer

- Stream
  - defined by
    - size of payload,
    - flavor of stream (valid tag, buffered, ...), and
    - processes being interconnected

- Signal
  - optional payload parameter
  - operations are post, wait

- Not all of C supported

Courtesy: Gokhale
/// PROCESS_FUN pe1_proc_run
/// INPUT input_stream
/// OUTPUT output_stream
/// PROCESS_FUN_BODY

SC_FLAG(tag);
  SC_REG(data, 32);

int i; int odata;

IF_SIM(printf("Process pe1_proc entered\n"));

SC_STREAM_OPEN(input_stream);
SC_STREAM_OPEN(output_stream);

while(SC_STREAM_EOS(input_stream) != SC_EOS) {
  SC_STREAM_READ(input_stream, data, tag);
  odata = SC_REG_GET_BITS_INT(data, 0, 32);
  odata |= 0xff00;
  SC_REG_SET_BITS_INT(data, 0, 32, odata);
  SC_STREAM_WRITE(output_stream, data, tag);
}

SC_STREAM_CLOSE(input_stream);
SC_STREAM_CLOSE(output_stream);

IF_SIM(printf("Process pe1_proc exiting\n"));
printf("Process pe1_proc exiting\n");

/// PROCESS_FUN_END

Process Declaration
Stream Declaration

Stream Operations
Processing Element Structure

- External Memory
  - Memory Interface
  - Signal Controller
- Datapath Module
  - Pipeline Control
  - Instruction Decode
  - Instruction Sequencer
- Stream Module
- Datapath
- Instruction Decode
- Instruction Sequencer
- Process 1
- Process 2
- Stream Module
Stream Hardware Components

- High bandwidth, synchronous communication
- Multiple protocols: “Valid” tag, buffered handshake
- Parameterized synthesizable modules
- Multiple channel mappings:
  - Intra-FPGA, Nearest neighbor, Crossbar, Host FIFO
PipeRench Architecture

- Many applications are primarily linear
  - Audio processing
  - Modified video processing
  - Filtering

- Consider a “striped” architecture which can be very heavily pipelined
  - Each stripe contains LUTs and flip flops
  - Datapath is bit-sliced
  - Similar to Garp/Chimaera but standalone

- Compiler initially converts dataflow application into a series of stripes

- Run-time dynamic reconfiguration of stripes if application is too big to fit in available hardware

Courtesy: Goldstein, Schmit
Stripped Architecture

- Same basic approach, pipelined communication, incremental modification
- Functions as a linear pipeline
- Each stripe is homogeneous to simplify computation
- Condition codes allow for some control flexibility
Piperench Internals

- Only multi-bit functional units used
- Very limited resources for interconnect to neighboring programming elements
- Place and route greatly simplified

![Diagram showing local and global routing with function units, shifts, registers, and MUXes.](image-url)
Since no loops and linear data flow used, first step is to perform topological sort.

Attempt to minimize critical paths by limiting NO-OP steps.

If too many trips needed, temporally as well as spatially pipeline.
PipeRench prototypes

- 3.6M transistors
- Implemented in a commercial 0.18 μ, 6 metal layer technology
- 125 MHz core speed (limited by control logic)
- 66 MHz I/O Speed
- 1.5V core, 3.3V I/O
Summary

• High-level is still not well understood for reconfigurable computing

• Difficult issue is parallel specification and verification

• Designers efficiency in RTL specification is quite high. Do we really need better high-level compilation?

• Hardware/software co-design an important issue that needs to be explored