Motivation

Many code sections are executed more efficiently in microprocessor: floating intensive codes, system calls, memory management functions, etc. To support codes containing these functions in FPGA, the FPGA should be able to call back to microprocessor as a master component.
Previous work

● Away from code coordination between CPU and FPGA
  ○ Handel-C, Impulse C
  ○ OCPIIP, AMBA

● Support nested and recursive only in hardware side
  ○ ASH (M. Budiu – ASPLOS ‘04), HybridThreads (E. Anderson-ERSA ‘07)
  ○ Do not allow hardware to call software

● Allows hardware to return back to software for software code execution
  ○ Comrade (H. Lange-FPL ‘07)
  ○ Do not support communication among compute units in FPGA

No work to support the cross calls between SW and HW without any limitation!
GCC2Verilog approach

- GCC2Verilog: A C-to-Verilog translator based on GCC compiler
  - Including a Verilog backend to generate Verilog code from GCC’s RTL

- Making hardware follows software calling convention
  - Software and hardware share one stack space.
    - Arguments passing through argument registers and stack.
  - Preserve software stack layout when performing calls in hardware side.

- Supporting:
  - Unlimited nesting calls in hardware including recursive calls.
  - Unlimited nesting cross calls between software and hardware.

Any hardware function in FPGA can be a master in the system!
Contents

- Compilation and Execution Model
- Address Resolution
- Additional Components
- Cross Calling Convention
- Experiment Results
- Conclusion
GCC2Verilog: Compilation & Execution Model

- **Code partitioning process:**
  - Divides codes into hardware and software sections
  - Prepares the address resolution

- **Compilation process:**
  - Compiles software code section into executable objects
  - Translates hardware code section into Verilog code and synthesizes them to HW bitstreams (HWIPs).

- **Execution process:**
  - Running SW executable code in a microprocessor & HWIPs in FPGA
  - The FPGA communicates with the host processor through a communication channel and memory.

http://compiler.korea.ac.kr
Address Resolution

- **Hardware address resolution:**
  - Assigning an hardware identification number *hwid* to each HWIP

- **Software address resolution:**
  - Static link: use the symbol table obtained an executable file to resolve software addresses at HLL-to-HDL translation.
  - Dynamic link:
    - Assign an identification number *swid* to each SW callee called from HW
    - Use an address_resolver() to obtain SW callee address at run time from *swid*

---

Original C code

```c
#pragma hardware function
int min (int a, int b, int c, int d, int e) {
  ...
}
int main(void) {
  int i, arr[10];
  int a, b, c, d, e;
#pragma hardware for
  for (i=0; i<10; i+=1) {
    arr[i] = min(a, b, c, d, e);
    printf("arr[%d] = %d\n", i, arr[i]);
  }
  ...
  return 0;
}
```

SW code section

```c
void (*func_ptr) (void);
void func_addr_resolver (int swid) {
  switch(swid) {
    case 1:
      func_ptr = &printf;
      break;
  }
}
int main (void) {
  int arr[10], a, b, c, d, e;
  wrapper (1, arr, a, b, c, d, e);
  ...
  return 0;
}
```

HW code section

```c
void hw_func_1(int* arr, int a, int b, int c, int d, int e) {
  int i;
  for (i=0; i<10; i+=1) {
    arr[i] = hw_func_2(i, b, c, d, e);
    printf("arr[%d] = %d\n", i, arr[i]);
  }
}

int hw_func_2(int a, int b, int c, int d, int e) {
  ...

SW address resolution in dynamic linking

http://compiler.korea.ac.kr
Additional Components

- HW controller:
  - Controls and schedules the execution between a processor and HWIPs

- SW/HW interface:
  - Provides a uniform interface to communicate with the host processor

- HW register set: set of registers for calls:
  - Argument registers
  - HW stack pointer
  - Link register

http://compiler.korea.ac.kr
Software Calls Hardware

1. The wrapper function passes arguments, and calls the HW callee
2. HW controller enables the HW callee
3. HW callee reads its arguments, and starts to execute
4. HW controller interrupts the host processor when the HW callee finishes.
5. The interrupt handler notifies the HW finishing to the wrapper.
Hardware Calls Software

1. HW caller passes arguments and notifies to the controller about the call.
2. HW controller interrupts the processor with SW callee ID.
3. The interrupt handler resolves the SW callee’s actual address from swid & the wrapper calls the function.

```plaintext
1. HW caller passes arguments and notifies to the controller about the call
2. HW controller interrupts the processor with SW callee ID
3. The interrupt handler resolves the SW callee’s actual address from swid & the wrapper calls the function.
```

![Diagram showing hardware calls software](image-url)
Hardware Calls Software

4. SW callee executes its code & returns to the wrapper when finish

HWIP's Argument 4
Pushed registers
Caller ID (return addr)
SW callee argument 4
Pushed registers
return addr

HWIP 1

Control unit
Datapath

HWIP N

Control unit
Datapath

SW/HW interface

HW controller

Stack space
Software Callee Returns to Hardware caller

5. The wrapper notifies to HW controller about SW finish
6. The HW caller is enabled again to continue its execution
Software calls hardware.

1. Processor
2. Hardware Interface
3. HW controller
4. SW/HW interface
5. Call + hwid = 2
6. Stack space
7. Enable
8. HWIP1's argument
9. HWIP2's argument
10. Pushed registers
11. Return addr
Hardware Calls Hardware

Processor

HWIP1

Control unit  
Datapath

Stack space

... 

HWIP1's argument
4

Pushed registers

HWIP2's argument
4

Pushed registers

Return addr

SW/HW interface

HW controller

return value

return addr

enable

finish
Experiment Result

- **Experiment setup**
  - Host processor: ARM922T
  - Benchmarks: EEMBC + factorial (recursion)

- **Calling overhead:**
  - Cross calls between SW and HW (exclude interrupting time)
    - Static link: 99 cycles
    - Dynamic link: 125 cycles
  - Calls among HWIPs:
    - Less than 5 cycles
# Experiment Result

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Number of calls</th>
<th>Call overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aifftr</td>
<td>300</td>
<td>3.52</td>
</tr>
<tr>
<td>aiifft</td>
<td>300</td>
<td>4.00</td>
</tr>
<tr>
<td>fft</td>
<td>100</td>
<td>2.71</td>
</tr>
<tr>
<td>bezier</td>
<td>20</td>
<td>0.11</td>
</tr>
<tr>
<td>idcitrn</td>
<td>600</td>
<td>4.62</td>
</tr>
<tr>
<td>rgbyiq</td>
<td>10</td>
<td>0.02</td>
</tr>
<tr>
<td>viterb</td>
<td>200</td>
<td>8.37</td>
</tr>
<tr>
<td>autcor</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>factorial</td>
<td>10</td>
<td>19.91</td>
</tr>
</tbody>
</table>

Call overhead including interrupt time
Conclusion

- Novel method to fully support cross calls among microprocessor and FPGA
  - Allowing FPGA to perform calls back to a microprocessor
  - Supporting unlimited nested and recursive calls in FPGA
- Reasonable cross calling overhead
- An importance step toward the full automatic translation of HLL to HDL
- Implemented a C-to-Verilog translator based on GCC compiler
Questions & Answers