Targeting XCore Resources from LLVM

Richard Osborne
Introduction

- XMOS
  - UK fabless semiconductor Company

- XCore processor
  - Aimed at embedded market
  - Hardware interfaces in software
    - Ethernet, USB 2.0, SPDIF, HDA, Uart, SPI, I2C, I2S, MIDI, CAN, SDRAM, VGA, …
XCore Processor

• Ports
  – Configurable I/O Engines
  – Interface between the processor and a collection of pins

• 8 hardware threads

• Inter-thread communication via channels

• Deterministic
XMOS Toolchain today

• C / C++ compiler
  – Based on llvm-gcc
  – Plain C

• XC Language and compiler
  – Compiler developed from scratch
  – Explicit parallelism
  – Operators for I/O and events
  – Compile time safety checks
Extending C with resources

• Goal: Efficiently target XCore hardware resources from C.

• Why?
  – Remove barrier of learning a new language
  – Evaluate feasibility of using LLVM backend for the XC compiler
  – Better C / XC interoperability
Choosing a frontend

• Choice of clang and llvm-gcc
• Decide to use clang
  – Adding XCore target trivial
    • < 100 lines of code
  – Design makes it easy to add language extensions
Resources

• Ports
  – Access collection of physical pins

• Timers
  – Access a free running 100MHz clock

• Chanends
  – Two chanends form a channel.
  – Allows communication between threads

• Resources share same basic usage model

• This talk will focus on Port I/O
Resource Identifiers

• 32 bit value which uniquely identifies a resource
• Used as operand to resource instructions
• Passed around as variable between functions
  - timer t;
  - port p;
  - chanend c;
I/O operations

• Basic operations
  – Input
    • Sample value from a port
    • Read the current time
    • Receive data from a channel
  – Output
    • Drive a value on a port
    • Output data to a channel
Port I/O Example

#include <res.h>

// Wait for value on pins to change
unsigned wait_for_transition(port p) {
    unsigned value = port_input(p);
    return port_input_ne(p, value);
}
Port I/O Example

```c
#include <res.h>

// Wait for value on pins to change
unsigned wait_for_transition(port p) {
    unsigned value = port_input(p);
    return port_input_ne(p, value);
}

# Unconditional input from port
setc res[r0], COND_NONE
in r1, res[r0]

# Conditional input on t
setc res[r0], COND_NE
setd res[r0], r1
in r0, res[r0]
```
I/O instructions

- **setc** instruction
  - Sets the port's condition register
- **setd** instruction
  - Sets the port's data register
- **in** instruction
  - Performs an input
  - Type of input depends on port state
  - Thread blocks until the input can be completed
Adding intrinsics

- **Clang builtins**
  - unsigned port_input(unsigned p);
  - unsigned port_input_ne(unsigned p, unsigned value);

- **LLVM intrinsics**
  - llvm.xcore.in
  - llvm.xcore.setc
  - llvm.xcore.setd

- **XCore backend**
  - llvm.xcore.in → in
  - llvm.xcore.setc → setc
  - llvm.xcore.setd → setd
State optimisations

• Port state is **persistent**
• Redundant state setting instructions can be eliminated

```assembly
call void @llvm.xcore.setc(i32 %p, i32 9)
call i32 @llvm.xcore.in(i32 %p)
call void @llvm.xcore.setc(i32 %p, i32 9)
call i32 @llvm.xcore.in(i32 %p)
```

• Loop invariant state can be hoisted

```assembly
loop:
call void @llvm.xcore.setc(i32 %p, i32 9)
call i32 @llvm.xcore.in(i32 %p)
br label %loop
```
Event driven programming

- An input waits until a single resource is ready
- Architecture also supports waiting for one out of a number resources becomes ready
Select statement

```c
select {
    case now = timer_input_after(t, time):
        break;
    case port_input_eq(p, 1):
        break;
}
```

- Case expression must be
  - input builtin
  - assignment with input builtin as RHS
- Thread pauses until one of the cases is ready
- Control jumps to the case label and that input is completed
Select statement

clre
setc res[r0], COND_EQ
setd res[r0], 1
eeu res[r0]
setv res[r0], case0
setc res[r1], COND_AFTER
setd res[r1], r2
setv res[r1], case1
eeu res[r1]
wai
eu

• The **eeu** instruction configures a resource to event when it is ready for input
• The **setv** instruction provides a vector to jump to when an event on the resource is taken
• The **waiteu** instructions pauses the thread until an event occurs

case0:
in r11, res[r0]
...
case 1:
in r11, res[r1]
Modelling select in IR

• Problem: It is not possible not take the address of a basic block in the LLVM IR
• Observation: The control flow for a select is similar to a switch
• i32 llvm.xcore.select(...)  
  – Takes a variable number of resources  
  – Returns a integer which is the position in the list of arguments of the resource which first become ready
call void @llvm.xcore.setc(i32 %t, i32 9)
call void @llvm.xcore.setd(i32 %d, i32 %timeout)
call void @llvm.xcore.setc(i32 %p, i32 17)
call void @llvm.xcore.setd(i32 %p, i32 1)
%0 = call i32 (...)* @llvm.xcore.select(i32 %t, i32 %p)
switch i32 %0, label %sel.epilog [  
  i32 0, label %case0  
  i32 1, label %case1
]

case0:
  %1 = call i32 @llvm.xcore.in(i32 %t)
  ...

case1:
  %2 = call i32 @llvm.xcore.in(i32 %p)
  ...
Lowering select

- Custom lowering implemented for switch with \texttt{llvm.xcore.select} argument

\begin{verbatim}
%0 = call i32 (...) @llvm.xcore.select(i32 %t, i32 %p)
switch i32 %0, label %sel.epilog [
  i32 0, label %sel.bb
  i32 1, label %sel.bb1
]
\end{verbatim}

\begin{verbatim}
c1re
setv res[r0], case0
setv res[r1], case1
waiteu
\end{verbatim}
Limitations

• Each resource must be listed as argument of llvm.xcore.select

• Impossible to select over an array of resources

```c
select {
  case (i = 0; i < n; i++) port_eq(p[i], x):
    break;
}
```
Wishlist

• Allow taking address of basic block in LLVM IR
• Allow indirect jump to basic block.
  – Possible targets listed on instruction
Conclusion

• Most I/O operations easily modelled using intrinsics

• Selects can be modelled as a switch implemented in hardware
  - Number of resources in the select must be known at compile time

• Current implementation does not support selecting on arrays of resources
Questions?