Software Defined Silicon

David May

Bristol University and XMOS

Introduction

We can build chips with hundreds of processors

We can build computers with millions of processors

We can support concurrent programming in *hardware*

We can define and build digital systems in *software*

Architecture

Regular, tiled implementation on chips, modules and boards

Scale from 1 to 1000 processors per chip

System interconnect with scalable throughput and low latency

Streamed (virtual circuit) or packetised communications

Architecture

High throughput, responsive input and output

Instruction set designed to support compiler optimisations

Power efficiency - compact programs and data, mobility

Energy efficiency - event driven systems

Interconnect

Supports multiple bidirectional links for each tile - a 500MHz processor can support several 100Mbyte/second streams

Exploit manufacturing processes with many layers of interconnect

Full bisection bandwidth can be achieved on silicon using crosspoint switches or multi-stage switches even for hundreds of links.

In some cases (eg modules and boards), low-dimensional grids are more practical.

Routing - modules and systems

Simple hardware operating on first few bits of message

Incoming bits compared with node address, bit-by-bit

If all pairs match, the node is the destination

If not, the bit number of the first non-matching pair is used to select an outgoing route via a lookup table

This is sufficient to perform efficient deadlock-free routing in all n-dimensional arrays - and many other structures.

Two-dimensional array example

node	entry	node	entry	node	entry	node	entry
0	rrdd	4	rldd	8	lrdd	12	lldd
1	rrdu	5	rldu	9	lrdu	13	lldu
2	rrud	6	rlud	10	lrud	14	llud
3	rruu	7	rluu	11	lruu	15	lluu

Each entry selects a right (r), left (l), up (u) or down (d) link.

Performance can be enhanced using a set of links on each path.

A set of links can be configured to provide multiple independent networks.

Interconnect protocol

Communication protocol provides *control* and *data* tokens.

- used to construct applications-optimised protocols.

A route is opened by a message *header* and closed by an *end-of-message* token.

The interconnect can be used under software control to

- establish virtual circuits to stream data or guarantee message latency
- perform dynamic packet routing by establishing and disconnecting circuits packet-by-packet.

Threads

Each processor includes *hardware* support for a number of threads, including:

- a set of registers for each thread
- a scheduler which dynamically selects which thread to execute
- a set of channels for communication with other threads
- a set of ports used for input and output
- a set of timers to control real-time execution
- a set of clock generators to enable synchronisation of the input-output with external time domains

Threads - use

Allow communications or input-output to progress together with processing.

Implement ‘hardware’ functions such as DMA controllers and specialised interfaces

Provide latency hiding by allowing some threads to continue whilst others are waiting for communication with remote tiles.

The set of threads in each tile can also be used to implement a kernel for a much larger set of software scheduled tasks.

Processor Instruction Set

Provide dedicated registers: *stack*, *data*, *constant pool* and *link*.

Provide 12 *operand* registers for everything else.

Key point: 11 bits can encode three operands
($12 \times 12 \times 12 < 2048$).

Compact program encoding using 16-bit and 32-bit instructions

Position independent code making code-mobility easy

Instructions

16-bit: 20 3-address: conventional arithmetic and logic
40 2-address: input-output and thread management
20 with 6-bit or 10-bit immediates: data access, branches
lots of 1-address and 0-address instructions

32-bit: less common instructions
instructions with up to 6 operands (for cryptography, DSP etc)
10-bit prefixes to extend range for jumps and stack offsets
lots of spare opcodes

Processor - Resources

Each processor manages physical resources: threads, synchronisers, channels, timers, locks and clock generators.

Threads *claim* and *free* resources using special instructions.

Resources interact directly with the thread scheduler and instructions such as inputs and outputs can potentially result in a process pausing until a resource is ready and then continuing.

Information about the state of a resource is available to the scheduler within a single processor cycle.

Process Scheduler

The thread scheduler maintains a set of runnable threads, *run*, from which it takes instructions in turn.

A thread is not in the *run* set when:

- its registers are being initialised prior to it being able to run.
- it is waiting to synchronise with another process before continuing or terminating.
- it has attempted an input but there is no data available.
- it has attempted an output but there is no room for the data.
- it is waiting for one of a number of events.

Thread Scheduler

Guarantees each of n threads at least $1/n$ processor cycles.

A chip with 128 processors each able to execute 8 threads can be used as if it were a chip with 1024 processors each operating at one eighth of the processor clock rate.

Shares a simple unified memory system between threads in a tile.

Each processor behaves as symmetric multiprocessor with 8 processors sharing a memory with no access collisions and with no caches needed.

Instruction Execution

Each thread has a short instruction buffer sufficient to hold at least four instructions.

Instructions are issued from the instruction buffers of the runnable threads in a round-robin manner.

Threads which are not in the *run* set are ignored.

Instruction fetch is performed within the execution pipeline, in the same way as data access.

Execution pipeline

Simple four stage pipeline:

1	decode	reg-write		
2		reg-read		
3	address	ALU1	resource-test	
4	read/write/fetch	ALU2	resource-access	schedule

At most *one* instruction per thread in the pipeline.

Use of Read/Write/Fetch stage

Any instruction which requires memory access performs it

Branch instructions fetch their branch target instructions - no branch penalty.

Any other instruction performs an instruction fetch for the thread (unless the buffer is full).

If a thread's buffer is empty when an instruction should be issued, a *no-op* is issued to fetch the next instruction.

Concurrency

Fast initiation and termination of threads

Fast barrier synchronisation - one instruction per thread

Compiler optimisation using barriers to remove join-fork pairs

Compiler optimisation of sequential programs using multiple threads (such as splitting an array operation into two half size ones)

Fork-join optimisation

```
while true
{ par { in(inchan,a) || out(outchan,b) };
  par { in(inchan,b) || out(outchan,a) }
}
```

```
par
{ while true
  { in(inchan,a); SYNC c; in(inchan,b); SYNC c }
|| while true
  { out(outchan,b); SYNC c; out(outchan,a); SYNC c }
}
```

Concurrent Software Components

```
while true
{ par { in(nextx) || in(nexty) || nextr := f(x, y) || out(r) };
  x, y, r := nextx, nexty, nextr
}
```

```
while true
{ par { in(nextx) || in(nexty) || nextr := f(x, y) || out(r) };
  par { move(nextx, x) || move(nexty, y) || move(nextr, r) }
}
```

Components can be composed to implement deterministic concurrent systems.

Concurrency and Synchronisation

Hardware synchronisers allow synchronisation to be performed at 1 instruction/thread

Instructions are provided to

- get a synchroniser
- get and attach new threads to a synchroniser
- free a synchroniser and all of its attached threads
- transfer data directly between the registers of two threads

These instructions interact with the scheduler via the synchronisers

Communication

Communication is performed using *channels*, which provide full-duplex data transfer between *channel ends*

The channel ends may be

- in the same processor
- in different processors on the same chip
- in processors on different chips

The channel-end identifiers can be used anywhere in a system

The channels provide a uniform method of communication throughout a system with multiple tiles or multiple chips.

Communication

Channel communication is implemented in hardware and does *not* involve memory accesses

This supports fine grained computations in which the number of communications is similar to the number of operations.

Within a tile, it is possible to use the channels to pass addresses.

Channels carry messages built from *tokens*

- data tokens are just bytes
- control tokens are used to encode communication protocols

Messages

A channel end is initialised with the destination channel end identifier.

When the first output is executed, the destination identifier is used as a message header to open a route through the interconnect.

Subsequent tokens follow this route, until an *end of message* control token is output.

Packet-based communication or virtual circuit communication can be *programmed*.

Channel Ends

A channel end can be used as a destination by any number of processes

They are served on a round-robin basis.

In this case the sender will normally send the identifier of its channel end which can be used to send a reply, or to establish bi-directional communication.

As the identifiers of channel ends can be used anywhere, they can themselves be communicated.

Synchronised Communications

As most messages consist of many individual data items, there is no need for all of the individual items to be acknowledged.

It is impossible to scale interconnect throughput unless communication is pipelined

This requires that the use of end-to-end synchronisations is minimised.

Synchronised communication is implemented by the receiver sending an acknowledgement to the sender, usually as a message consisting of a header and an end-of-message token.

Compound Communications

A convenient way to express sequences of communications on the same channel is with a *compound communication*.

```
proc inarray(chan c, []int a) is
?{ for i = 0 for 10 do c ? a[i] ?}

proc outarray(chan c, []int a) is
!{ for i = 0 for 10 do c ! a[i] !}
```

The synchronisations at the end of each of these compound communications ensure that each compound output is matched by exactly one compound input.

Ports, Input and Output

Ports provide interfaces to physical pins.

Inputs and outputs using ports provide

- direct access to the pins
- accesses synchronised with a clock
- accesses timed under program control

An input can be delayed until a specified condition is met

- the time at which the condition is met can be *timestamped*

Timers and Clocks

Each tile has a free-running clock and a set of timers which can be used to read the current time or to wait until a specified time.

Input and output operations can be synchronised with an internally generated clock or an externally supplied clock.

When an output port is driven from a clock, the data on the pin(s) changes state synchronously with the clock.

If several ports are driven from the same clock, they will appear to operate as a single port.

Time domains

The threads executed by a processor can handle external devices at several different rates determined by clocks supplied externally or generated internally.

The threads decouple the internal timing of input and output program execution from the operation of the input and output interfaces.

The processor can operate using its own clock, or could potentially be asynchronous.

Ports, Input and Output example

```
proc linkin(port in_0, in_1, ack, int token) is
var state_0, state_1, state_ack;
{ state_0 := 0; state_1 := 0; state_ack = 0; token := 0;
  for bitcount = 0 for 10 do
  { select
    { case in_0 ?= ¬state_0: state_0 => token := token>>1
      case in_1 ?= ¬state_1: state_1 => token:=(token>>1)|512
    };
    ack ! state_ack; state_ack := ¬state_ack
  }
}
```

Timed ports example

```
proc uartin(port uin, byte b) is
{ var starttime;
  in ?= 0 at starttime;
  sampletime := starttime + bittime/2;
  for i = 0 for 8
   t := t + bittime; (uin at t) ? >> b ;
  (uin at (t + bittime)) ? nil
}
```

Events

A thread can wait for an event from one of a set of channels, ports or timers

An *entry point* is set for each resource by a *setvector* instruction; event generation for each resource can then be enabled and disabled using *enable* instructions

A *wait* instruction is used to wait until an event transfers control directly to its associated entry point.

All of the events enabled by a thread can be disabled by a single *clear events* instruction.

Events - optimisations

Optimise repeated event-handling in inner loops where a thread is operating as a programmable state machine - the events are often handled by (very) short instruction sequences

Move setting of vectors and other invariants outside the loop

Provide conditional versions of *event enable* instructions to optimise enabling and disabling

Provide conditional versions of *wait* to replace the loop-closing branch

Events vs. Interrupts

A thread can be dedicated to handling an individual event or to responding to multiple events.

The data needed to handle each event have been initialised prior to waiting, and will be instantly available when the event occurs.

This is in sharp contrast to an interrupt-based system in which context must be saved and the interrupt handler context restored prior to entering it - and the converse when exiting.

Summary

Concurrent programming can be efficiently supported in *hardware* using tiled multicore chips.

They enable systems to be defined and built using *software*.

Each thread can be used

- to run conventional sequential programs
- as a component of a concurrent computer
- as a hardware emulation engine or input-output controller

Event-driven hardware and software enable energy efficient systems.

XMOS XS1 tile

Processor	500 MHz; 8 threads
SRAM	64k bytes
Synchronisers	7
Timers	10
Channel ends	32
Ports	1,4,8,16,32-bit
Links	4 at 100Mbyte/second

Prototype has 4 tiles communicating via a fully-connected switch