LG4: Electronic System Level (ESL) Modelling.

Topic: Electronic System Level (ESL) Modelling.

- LG4.1 - ESL Motivation 1: Architectural Exploration
- LG4.2 - ESL Example H/W Protocol: 4P Handhsake
- LG4.3 - ESL What is a transaction ?
- LG4.4 - ESL Adding Timing Annotations
- LG4.5 - ESL TLM in SystemC: TLM 1.0
- LG4.6 - SoC Component, TLM Form Example
- LG4.7 - ESL TLM in SystemC: TLM 2.0
- LG4.8 - ESL Timing Models
- LG4.9 - ESL Approximate Timing
- LG4.10 - ESL Loose Timing

LG4.1 - ESL Motivation 1: Architectural Exploration

Rapid prototyping of a SoC architecture.

Evaluate bus bandwidth and memory size use.

Get a feel for performance (accurate or loose).

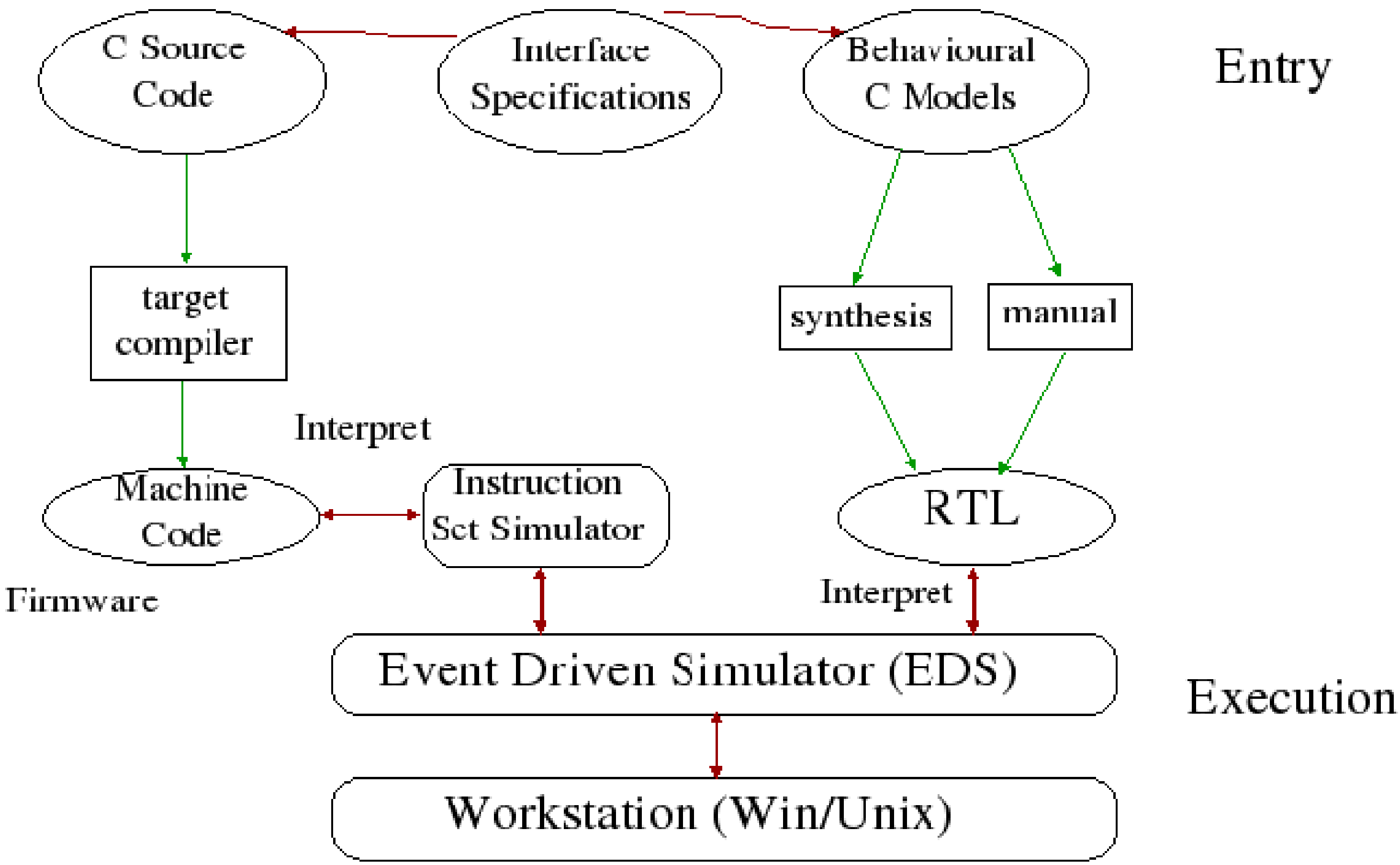
Generate some high-level behavioural models of major components.

Future topic: Write some assertions about them (ABD).

Future world: Behavioural models synthesised to actual system.

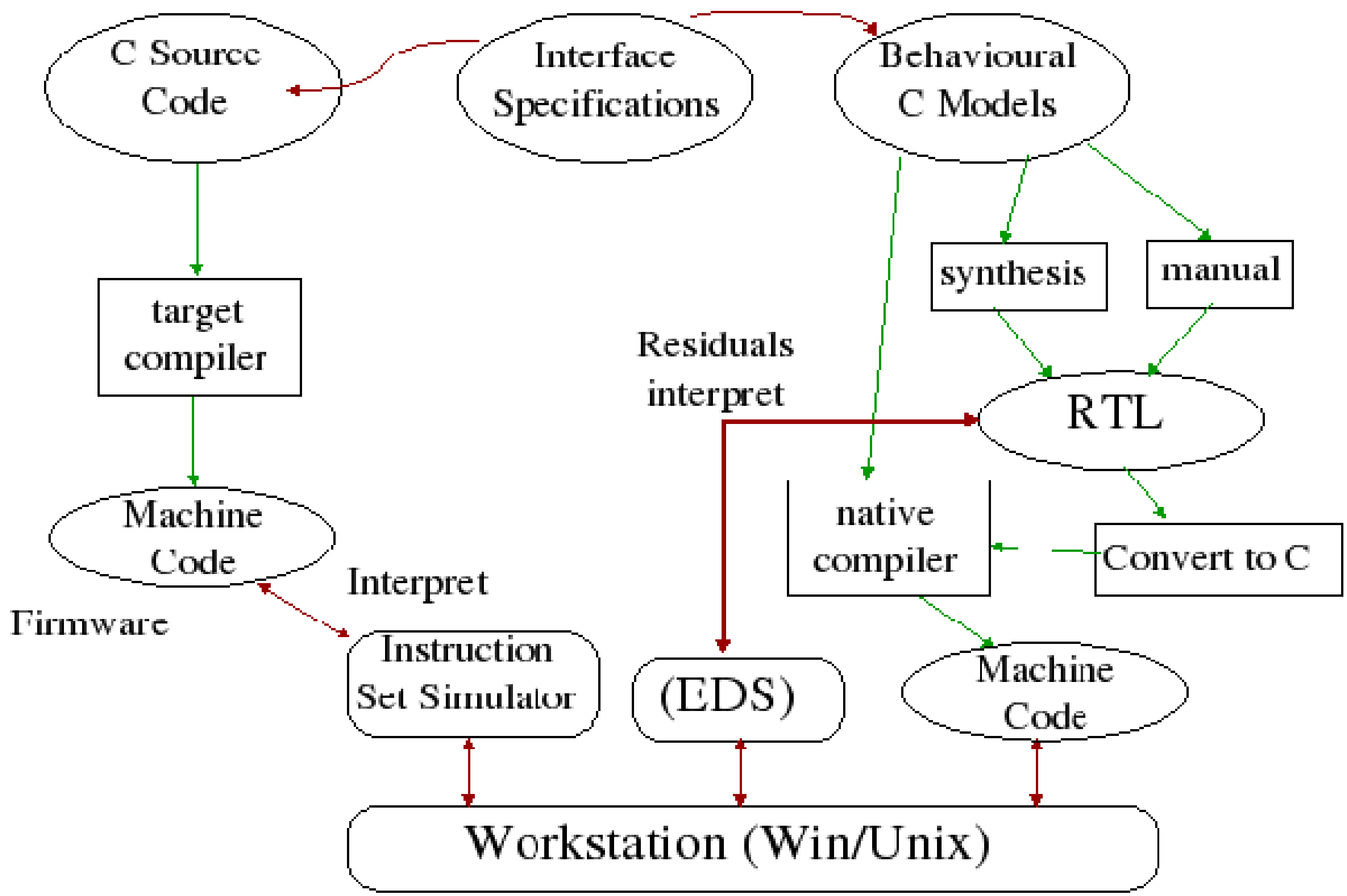
Near future world: Behavioural models used in test benches and hybrid models.

Pre-Electronic System Level (Pre-ESL)



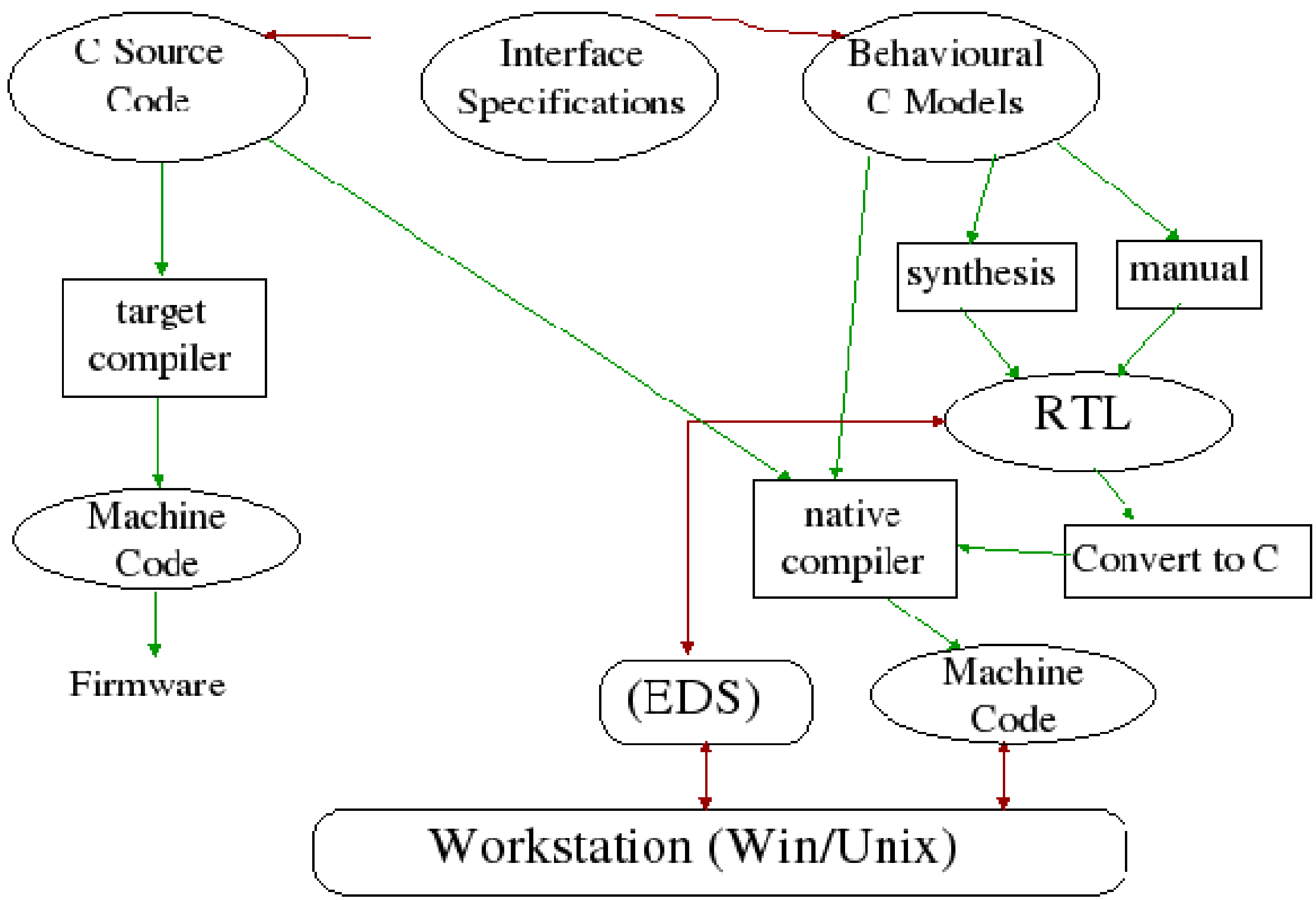
ISS slowly interprets binary machine code. RTL simulator slowly simulates hardware devices.

Pre-ESL 2: Using native C compiler



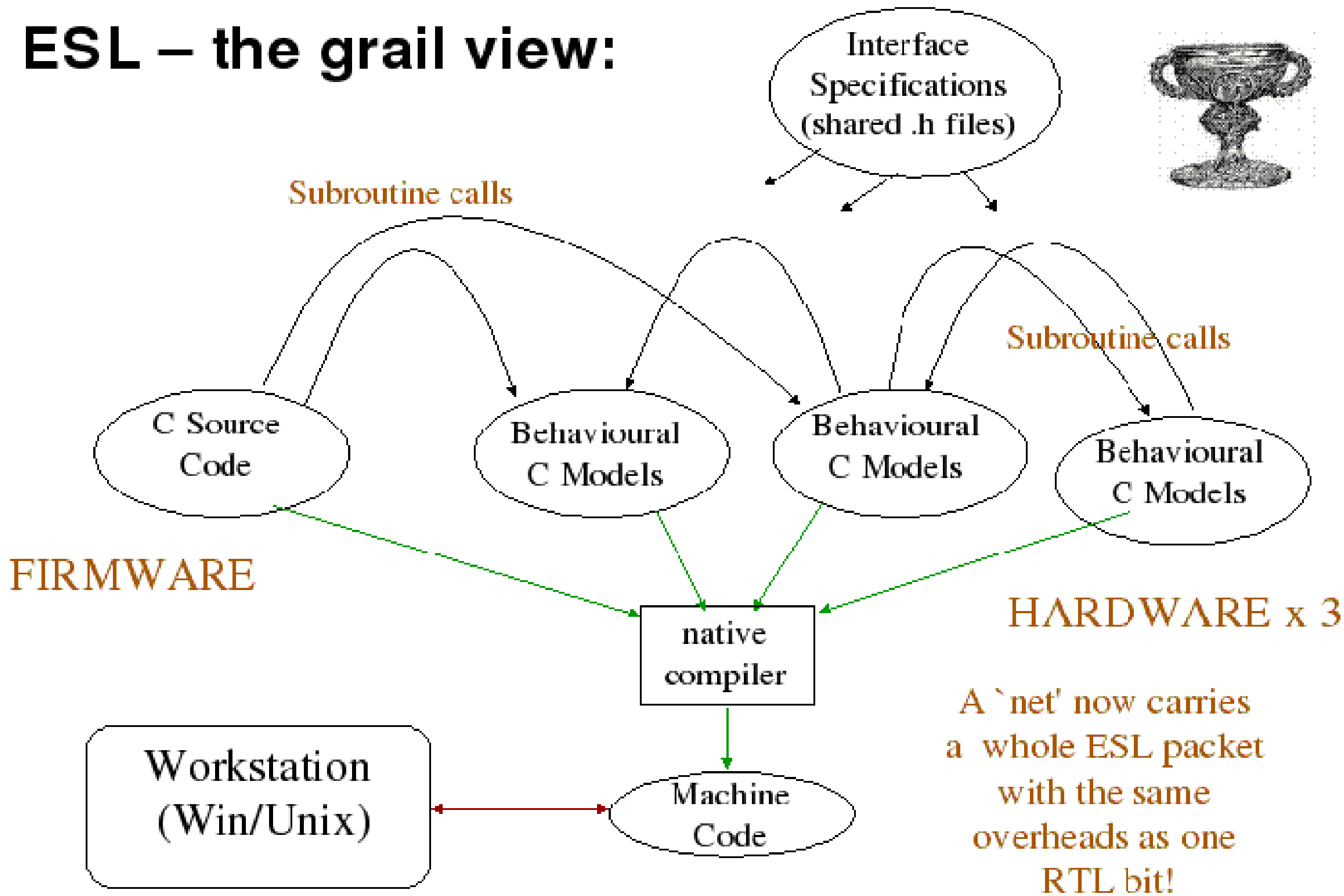
Hardware devices compiled to cycle-accurate C models: increased performance.

Pre-ESL 3: Avoiding ISS



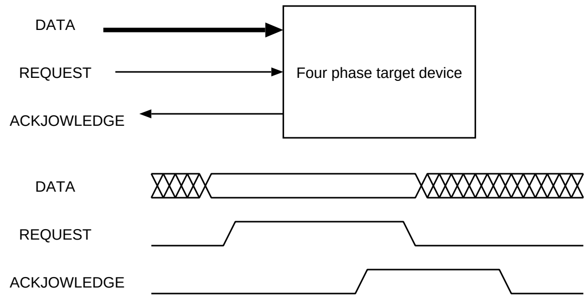
Firmware cross-compiled for workstation processor: avoids ISS.

ESL – the grail view:



ESL: Behavioural models used as simulation models.

LG4.2 : Example H/W Protocol: 4P Handhsake




```
putbyte(char d)
{
  wait_until(!ack);
  data = d;
  settle();
  req = 1;
  wait_until(ack);
  req = 0;
}
```

```
char getbyte()
{
  wait_until(req);
  char r = data;
  ack = 1;
  wait_until(!req);
  ack = 0;
  return r;
}
```

Example, untimed, blocking transactor code: converts from ESL to pin-level.

A commonly used asynchronous, simplex protocol, with flow control.

See full SystemC versions in additional material.

LG4.3 - ESL What is a transaction ?

Computer science : a transaction has atomicity, with commit or rollback.

ESL: A thread from one component executes a method on another.

```
bool putbyte_nb_start(char d)
{
  if (ack) return false;
  data = d;
  settle(); // hmmm!
  req = 1;
  return true;
}

bool putbyte_nb_end(char d)
{
  if (!ack) return false;
  req = 0;
  return true;
}
```

```
bool getbyte_nb_start(char &r)
{
  if (!req) return false;
  r = data;
  ack = 1;
  return true;
}

bool getbyte_nb_end()
{
  if (req) return false;
  ack = 0;
  return true;
}
```

Blocking: Hardware flow control signals implied by thread's call and return.

Non-blocking: Success status returned and caller must poll/retry.

In SystemC: blocking requires an SC_THREAD, whereas non-blocking can use an SC_METHOD.

Choice: a matter of style ? (TLM 2.0 sockets will even automatically map.)

Non-blocking enables finer-grained concurrency and closer to cycle-accurate timing results.

LG4.4 - ESL Adding Timing Annotations

```
putbyte(char d, sc_time &dt)
{
  ...
  dt += sc_time(140, SC_NS);
}
```

```
char getbyte(sc_time &dt)
{
  do { wait(0, SC_NS); } until(req);
  char r = data;
  ack = 1;
  do { wait(0, SC_NS); } until(!req);
  ack = 0;
  return r;
}
```

```
bool putbyte_nb_start(char d, sc_time &dt)
{
  if (ack) return false;
  data = d;
  settle(); // hmmm!
  req = 1;
  return true;
}

bool putbyte_nb_end(char d, sc_time &dt)
{
  if (!ack) return false;
  req = 0;
  return true;
}
```

```
bool getbyte_nb_start(char &r,
                      sc_time &dt)
{
  if (!req) return false;
  r = data;
  ack = 1;
  return true;
}

bool getbyte_nb_end(sc_time &dt)
{
  if (req) return false;
  ack = 0;
  return true;
}
```

What do we do with `sc_time` in each case?

LG4.5 - ESL TLM in SystemC: TLM 1.0

TLM1.0 standard used conventional C++ concepts of multiple inheritance.

An SC_MODULE that implements an interface just inherits it.

The `sc_port` and `sc_export` constructs are used to wire TLM ports together.

Problem: no standardised structure for payloads.

Problem: no standardised timing annotation mechanism.

Problem: how to have multiple TLM ports on a component with same interface: e.g. a packet router.

NB: Full exam credit gained using any of TLM1.0 or TLM2.0 styles or even your own psudo code.

LG4.6 - SoC Component, TLM Form Example.

Example: a one-channel DMA controller:

```
// Bus slave side, operand registers
uint32 src, dest, length;
bool busy, int_enable;

uint32 status() { return (busy << 31)
          | (int_enable << 30); }

uint32 slave_read(int a)
{
  return (a==0)? src: (a==4) ? dest:
   (a==8) ? (length) : status();
}
void slave_write(int a, uint32 d)
{
   if (a==0) src=d;
   else if (a==4) dest=d;
   else if (a==8) length = d;
   else if (a==12)
   { busy = d >> 31;
     int_enable = d >> 30; }
}
```

```
// Bus mastering side
while(1)
{
  waituntil(busy);
  while (length-- > 0)
    mem.write(dest++, mem.read(src++));
  busy = 0;
}
```

Like to make interrupt output with an RTL-like continuous assignment:

```
interrupt = int_enable & !busy;
```

But this will need a thread to run it

LG4.7 - ESL TLM in SystemC: TLM 2.0

TLM2.0 (July 2008) defines the **Generic Payload**

Also defines memory/garbage ownership and transport primitives with timing.

```
trans->set_command(tlm::TLM_WRITE_COMMAND);
trans->set_address(addr);
trans->set_data_ptr(reinterpret_cast<unsigned char*>(&data));
trans->set_data_length(4);
trans->set_streaming_width(4);
trans->set_byte_enable_ptr(0);
trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

socket->b_transport(*trans, delay);
```

Other standard payloads (e.g. 802.3 frame or audio sample) might be expected soon ?

LG4.8 ESL - Timing Models

The SystemC kernel time_stamp ?

```
cout << ``Time now is : `` << simcontext()->time_stamp() << `` \n'';
```

Coarse processing granularity: high-speed simulation.

How often must we re-enter the SystemC kernel ?

Can we do a large number of ISS instructions at a time ?

- Cycle Accurate - Every clock tick.
- Approximately Timed - Every TLM call and return.
- Loosely Timed - When we need a result from another component.
- Untimed - Never ?

LG4.9 - ESL Approximate Timing

Approximately-timed

Supported directly by the non-blocking transport interface.

Appropriate for architectural exploration and performance analysis.

Processes typically need to run in lockstep with SystemC simulation time.

Delays annotated onto process interactions are implemented using `wait` calls.

Multiple transaction phases with timing points.

Four phases of the base protocol, namely BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP.

Uses a backward path (see additional material).

LG4.10 - ESL Loose Timing

Loosely-timed (temporal decoupling)

The loosely-timed coding style makes use of the blocking transport interface.

Two timing points per transaction:call and return.

Loosely-timed is appropriate for software development.

It supports modelling of timers and interrupts, sufficient to boot an operating system.

Simulation time still exists, but processes may be temporally decoupled from simulation time.

Each process keeps a tally of how far it has run ahead of simulation time, and may yield because it reaches an explicit synchronization point or because it has consumed its **time quantum**.

Time quantum start is current simulation time as returned by sc_time_stamp().

Delays in the b_transport and nb_transport calls methods are local time offsets defined relative to the start of the time quantum.

Large time quantum: fast simulation.

Small time quantum: transaction order interleaving is more accurate.