CprE 588
Embedded Computer Systems

Prof. Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University

Lecture #4 – Introduction to SpecC
Introduction

- System-on-Chip (SOC) design
- Increase of design complexity
- Move to higher levels of abstraction

The SpecC Model

- **Traditional model**
  - Processes and signals
  - Mixture of computation and communication
  - Automatic replacement impossible

- **SpecC model**
  - Behaviors and channels
  - Separation of computation and communication
  - Plug-and-play
The SpecC Model (cont.)

- **Specification model**
- **Exploration model**

- Computation in behaviors
- Communication in channels

- **Implementation model**

- Channel disappears
- Communication inlined into behaviors
- Wires exposed
System-Level Language Goals

- **Executability**
  - Validation through simulation
- **Synthesizability**
  - Implementation in HW and/or SW
  - Support for IP reuse
- **Modularity**
  - Hierarchical composition
  - Separation of concepts
- **Completeness**
  - Support for all concepts found in embedded systems
- **Orthogonality**
  - Orthogonal constructs for orthogonal concepts
  - Minimality
- **Simplicity**
# System-Level Language Requirements

<table>
<thead>
<tr>
<th>Behavioral hierarchy</th>
<th>C</th>
<th>C++</th>
<th>Java</th>
<th>VHDL</th>
<th>Verilog</th>
<th>HardwareC</th>
<th>Statecharts</th>
<th>SpecCharts</th>
<th>SpecC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Structural hierarchy</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Supported</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Concurrency</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Supported</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Synchronization</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Supported</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Exception handling</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Supported</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Timing</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>State transitions</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Composite data types</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

- **Not supported**
- **Partially supported**
- **Supported**

---

Feb 3-5, 2009  
CprE 588 – Embedded Computer Systems  
Lect-04.6
The SpecC Language

• Foundation: ANSI-C
  • Software requirements are fully covered
  • SpecC is a true superset of ANSI-C
  • Every C program is a SpecC program
  • Leverage of large set of existing programs
  • Well-known
  • Well-established

• SpecC has extensions needed for hardware
  • Minimal, orthogonal set of concepts
  • Minimal, orthogonal set of constructs

• SpecC is a real language
  • Not just a class library
The SpecC Language (cont.)

• ANSI-C
  • Program is set of functions
  • Execution starts from function main()

• SpecC
  • Program is set of behaviors, channels, and interfaces
  • Execution starts from behavior Main.main()
The SpecC Language (cont.)

• SpecC types
  • Support for all ANSI-C types
    • predefined types (int, float, double, …)
    • composite types (arrays, pointers)
    • user-defined types (struct, union, enum)
  • Boolean type:
    Explicit support of truth values
    • bool b1 = true;
    • bool b2 = false;
  • Bit vector type:
    Explicit support of bit vectors of arbitrary length
    • bit[15:0] bv = 1111000011110000b;
  • Event type:
    Support for synchronization
    • event e;
The SpecC Language (cont.)

- Bit vector type
  - signed or unsigned
  - arbitrary length
- standard operators
  - logical operations
  - arithmetic operations
  - comparison operations
  - type conversion
  - type promotion
- concatenation operator
  - \texttt{a @ b}
- slice operator
  - \texttt{a[l:r]}

```c
typedef bit[7:0] byte; // type definition
byte a;
unsigned bit[16] b;

{
    bit[31:0] r;
    a = 11001100b;           // constant
    b = 1111000011110000ub;  // assignment
    b[7:0] = a;               // sliced access
    b = d[31:16];

    if (b[15])               // single bit
        b[15] = 0b;          // access
    r = a @ d[11:0] @ c      // concatenation
        @ 11110000b;
    a = ~(a & 11110000);    // logical op.
    r += 42 + 3*a;          // arithmetic op.

    return r;
}
```
The SpecC Language (cont.)

- Basic structure
  - Top behavior
  - Child behaviors
  - Channels
  - Interfaces
  - Variables (wires)
  - Ports
The SpecC Language (cont.)

• Basic structure

```
interface I1
{
    bit[63:0] Read(void);
    void Write(bit[63:0]);
};

channel C1 implements I1;

behavior B1(in int, I1, out int);

behavior B(in int p1, out int p2)
{
    int v1;
    C1 c1;
    B1 b1(p1, c1, v1),
        b2(v1, c1, p2);

    void main(void)
    {
        par {
            b1.main();
            b2.main();
        }
    }
};
```
The SpecC Language (cont.)

- Typical test bench
  - Top-level behavior: Main
  - Stimulator provides test vectors
  - Design unit under test
  - Monitor observes and checks outputs
The SpecC Language (cont.)

- Behavioral hierarchy

Sequential execution

```
behavior B_seq
{
    B b1, b2, b3;
}
```

void main(void)
{
    b1.main();
    b2.main();
    b3.main();
}

FSM execution

```
behavior B_fsm
{
    B b1, b2, b3,
    b4, b5, b6;
    void main(void)
    { fsm { b1:{...} b2:{...} ... } }
}
```

Concurrent execution

```
behavior B_par
{
    B b1, b2, b3;
    void main(void)
    { par{b1.main(); b2.main(); b3.main(); } }
}
```

Pipelined execution

```
behavior B_pipe
{
    B b1, b2, b3;
    void main(void)
    { pipe(b1.main(); b2.main(); b3.main(); ) }
}
```
SpecC – FSM Support

- Finite State Machine (FSM)
  - Explicit state transitions
    - triple $<$ current_state, condition, next_state $>$
    - \textbf{fsm} \{ current_state: \{ if condition goto next_state \} \} ...
  - Moore-type FSM
  - Mealy-type FSM

```c
behavior B_FSM(in int a, in int b) {
    B b1, b2, b3;

    void main(void) {
        fsm { b1: { if (b<0) break;
                     if (b==0) goto b1;
                     if (b>0) goto b2; }
            b2: { if (a>0) goto b1; }
            b3: { if (a>b) goto b1; }
    }
    }
}
```

![Diagram](image.png)
SpecC – Pipelining Support

• Pipeline
  • Explicit execution in pipeline fashion
    • `pipe { <instance_list> };`

```c
behavior Pipeline {
  Stage1 b1;
  Stage2 b2;
  Stage3 b3;

  void main(void) {
    pipe {
      b1.main();
      b2.main();
      b3.main();
    }
  }
}
```
SpecC – Pipelining Support (cont.)

- Pipeline
  - Explicit execution in pipeline fashion
    - `pipe { <instance_list> }`;
    - `pipe (<init>; <cond>; <incr>) { ... }`
SpecC – Pipelining Support (cont.)

- Pipeline
  - Explicit execution in pipeline fashion
    - `pipe { <instance_list> }`;
    - `pipe (<init>; <cond>; <incr>) { ... }
  - Support for automatic buffering

```plaintext
behavior Pipeline
{
  int v1;
  int v2;
  int v3;

  Stage1 b1(v1, v2);
  Stage2 b2(v2, v3);
  Stage3 b3(v3, v1);

  void main(void)
  {
    int i;
    pipe(i=0; i<10; i++)
      { b1.main();
        b2.main();
        b3.main();
      }
  }
};
```
SpecC – Pipelining Support (cont.)

- Pipeline
  - Explicit execution in pipeline fashion
    - `pipe { <instance_list> };`
    - `pipe (<init>; <cond>; <incr>) { … }`
  - Support for automatic buffering
    - `piped […] <type> <variable_list>;

```c
behavior Pipeline
{
    piped piped int v1;
    piped int v2;
    piped int v3;
    Stage1 b1(v1, v2);
    Stage2 b2(v2, v3);
    Stage3 b3(v3, v1);

    void main(void)
    {
        int i;
        pipe (i=0; i<10; i++)
        {
            b1.main();
            b2.main();
            b3.main();
        }
    }
}
```
SpecC – Communication

• Communication
  • via shared variable
  • via virtual channel
  • via hierarchical channel
SpecC – Synchronization

- Synchronization
  - Event type
    - `event <event_list>;`
  - Synchronization primitives
    - `wait <event_list>;`
    - `notify <event_list>;`
    - `notifyone <event_list>;`

```cpp
behavior S(out event Req, 
  out float Data, 
  in event Ack)
{
    float X;
    void main(void)
    {
      ... 
      Data = X;
      notify Req;
      wait Ack;
      ...
    }
};
```

```cpp
behavior R(in event Req, 
  in float Data, 
  out event Ack)
{
    float Y;
    void main(void)
    {
      ...
      wait Req;
      Y = Data;
      notify Ack;
      ...
    }
};
```
SpecC – Communication (cont.)

- Communication
  - Interface class
    - `interface <name> { <declarations> };`
  - Channel class
    - `channel <name> implements <interfaces> { <implementations> };`

```c
interface IS
{
    void Send(float);
};
interface IR
{
    float Receive(void);
};

channel C implements IS, IR
{
    event Req;
    float Data;
    event Ack;

    void Send(float X)
    {
        Data = X;
        notify Req;
        wait Ack;
    }

    float Receive(void)
    {
        float Y;
        wait Req;
        Y = Data;
        notify Ack;
        return Y;
    }
};
```
Hierarchical channel
- Virtual channel implemented by standard bus protocol
  - example: PCI bus
SpecC – Miscellaneous (cont.)

- Exception handling

```c
behavior B1(in event e1, in event e2)
{
    B b, a1, a2;

    void main(void)
    {
        try { b.main(); }
        trap (e1) { a1.main(); }
        trap (e2) { a2.main(); }
    }
}

behavior B2(in event e1, in event e2)
{
    B b, i1, i2;

    void main(void)
    {
        try { b.main(); }
        interrupt (e1) { i1.main(); }
        interrupt (e2) { i2.main(); }
    }
}
```
SpecC – Timing

• Timing
  • Exact timing
    • `waitfor <delay>;'
SpecC – Timing (cont.)

- Timing
  - Exact timing
    - `waitfor <delay>;`
  - Timing constraints
    - `do { <actions> } timing {<constraints>}`

Example: SRAM read protocol

```c
bit[7:0] Read_SRAM(bit[15:0] a) {
  bit[7:0] d;
  do { t1: {ABus = a; } 
    t2: {RMode = 1; 
          WMode = 0; } 
    t3: { } 
    t4: {d = Dbus; } 
    t5: {ABus = 0; } 
    t6: {RMode = 0; 
          WMode = 0; } 
    t7: { } 
  } 
  timing { range(t1; t2; 0; ); 
            range(t1; t3; 10; 20); 
            range(t2; t3; 10; 20); 
            range(t3; t4; 0; ); 
            range(t4; t5; 0; ); 
            range(t5; t7; 10; 20); 
            range(t6; t7; 5; 10); 
  } 
  return(d); 
}
```
SpecC – Timing (cont.)

- **Timing**
  - **Exact timing**
    - `waitfor <delay>;`
  - **Timing constraints**
    - `do { <actions> }
      timing {<constraints>} `

**Example: SRAM read protocol**

```plaintext
bit[7:0] Read_SRAM(bit[15:0] a)
{
  bit[7:0] d;

  do { t1: {ABus = a; waitfor( 2);}  
     t2: {RMode = 1;  
          WMode = 0; waitfor(12);}  
     t3: { waitfor( 5);}  
     t4: {d = Dbus; waitfor( 5);}  
     t5: {ABus = 0; waitfor( 2);}  
     t6: {RMode = 0;  
          WMode = 0; waitfor(10);}  
     t7: { }
       
     timing { range(t1; t2;  0;    );  
             range(t1; t3; 10; 20);  
             range(t2; t3; 10; 20);  
             range(t3; t4;  0;    );  
             range(t4; t5;  0;     );  
             range(t5; t7; 10; 20);  
             range(t6; t7;  5; 10);  
             }
       
     return(d);
  }
```

Implementation 1
SpecC – Timing (cont.)

- Timing
  - Exact timing
    - `waitfor <delay>;`
  - Timing constraints
    - `do { <actions> } timing {<constrants>}`

Example: SRAM read protocol

```
bit[7:0] Read_SRAM(bit[15:0] a)
{
    bit[7:0] d;       // ASAP Schedule
    do {
        t1: {ABus = a;  }
        t2: {RMode = 1;
              WMode = 0; waitfor(10);}
        t3: { }
        t4: {d = Dbus;  }
        t5: {ABus = 0;  }
        t6: {RMode = 0;
              WMode = 0; waitfor(10);}
        t7: { }
    }
    timing {
        range(t1; t2;  0;   );
        range(t1; t3; 10; 20);
        range(t2; t3; 10; 20);
        range(t3; t4;  0;   );
        range(t4; t5;  0;   );
        range(t5; t7; 10; 20);
        range(t6; t7;  5; 10);
    }
    return(d);
}
```
SpecC – Miscellaneous (cont.)

- Library support
  - Import of precompiled SpecC code
    - `import <component_name>;
  - Automatic handling of multiple inclusion
    - no need to use `#ifdef - #endif` around included files
  - Visible to the compiler/synthesizer
    - not inline-expanded by preprocessor
    - simplifies reuse of IP components

```c
// MyDesign.sc

#include <stdio.h>
#include <stdlib.h>

import "Interfaces/I1";
import "Channels/PCI_Bus";
import "Components/MPEG-2";
...
```
Persistent annotation

- Attachment of a key-value pair
  - globally to the design, i.e. `note <key> = <value>;
  - locally to any symbol, i.e. `note <symbol>.<key> = <value>;

- Visible to the compiler/synthesizer
  - eliminates need for pragmas
  - allows easy data exchange among tools

```c
/* comment, not persistent */

// global annotations
note Author = "Rainer Doemer";
note Date   = "Fri Feb 23 23:59:59 PST 2001";

behavior CPU(in event CLK, in event RST, ...) {
  // local annotations
  note MaxClockFrequency = 800 * 1e6;
  note CLK.IsSystemClock = true;
  note RST.IsSystemReset = true;
...}
```
Summary

- SpecC model
  - PSM model of computation
  - Separation of communication and computation
  - Hierarchical network of behaviors and channels
  - Plug-and-play
- SpecC language
  - True superset of ANSI-C
    - ANSI-C plus extensions for HW-design
  - Support of all concepts needed in system design
    - Structural and behavioral hierarchy
    - Concurrency
    - State transitions
    - Communication
    - Synchronization
    - Exception handling
    - Timing
Conclusion

• SpecC language
  • Executable and synthesizable
  • Precise coverage of system language requirements
  • Orthogonal constructs for orthogonal concepts

• Impact
  • Adoption of SpecC in industry and academia
  • SpecC Open Technology Consortium (STOC)

• Future
  • Standardization effort in progress by STOC
  • Improvement with your participation