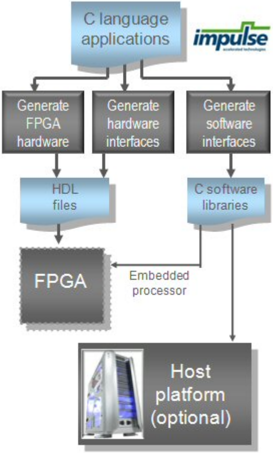
Introduction to Impulse C

Hardware and Software Design for Cryptographic Applications

April 4, 2013

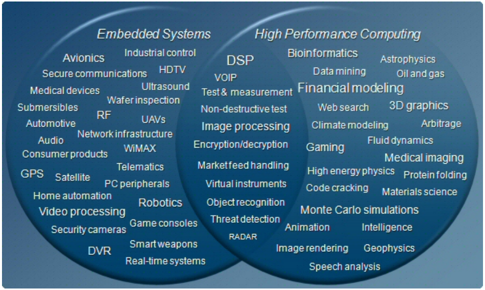
What is Impulse C?

- C-language for FPGA programming targeting embedded and HPC applications
- A software-to-hardware compiler
  - Optimizes C code for parallelism
  - Generates HDL, ready for FPGA synthesis
  - Generates hardware/software interfaces
- Purpose: Describe hardware accelerators using C

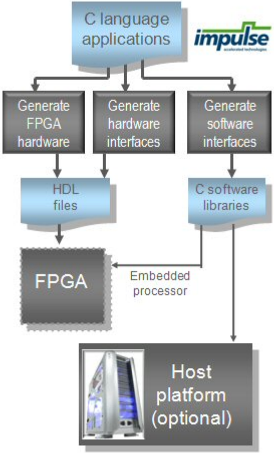


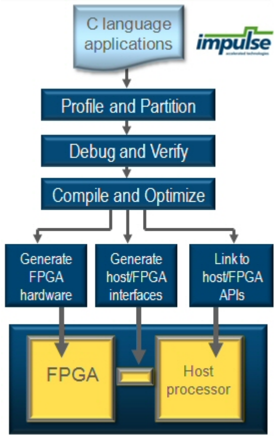
Why use Impulse C?

- Reduce application development time
- Reduce cost of entry
- Reduce project costs
- Provide “good enough” alternative to hardware design prototyping

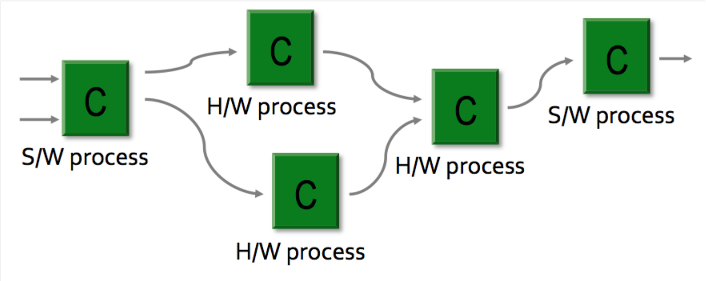


Impulse C Design Flow



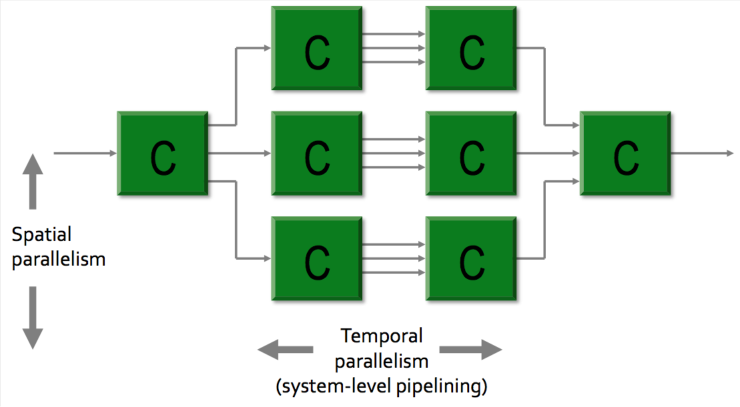


Programming Model



- Communicating C-language processes using an extension of standard ANSI C
- Buffered communication channels to implement data streams
- Simplified expression of parallel algorithms using well-defined data communication, message passing, and synchronization mechanisms

Parallelism via Multiple Processes



Impulse C Language Notes

- Only a subset of C can be compiled to hardware
  - No recursion
  - No pointers other than ones that can be resolved at compile-time
  - Limited support for complex data types
- Data types, compiler directives, and functions add extra functionality
  - Multiple-process parallelism (e.g. `co_process`)
  - Non-standard data types (e.g. `co_uint27`)
  - `CO` compiler directives
- Optimization pragmas allow for control of synthesized hardware
  - `CO UNROLL`
  - `CO PIPELINE`
  - `CO SET stageDelay`
  - ...

Signed and Unsigned Data Types

Impulse C provides similar data type and width flexibility as in popular HDLs

`co_int1`	1-bit signed integer
`co_int7`	7-bit signed integer
`co_uint16`	16-bit unsigned integer
`co_uint24`	24-bit unsigned integer
`co_int32`	32-bit signed integer
`co_uint64`	64-bit unsigned integer

Processes

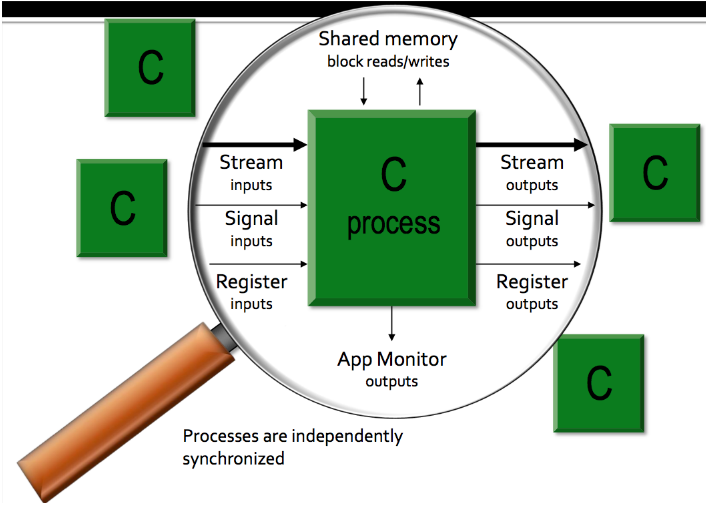
- Fundamental units of computation in an Impulse C application
- Executed as independent synchronized units of code on the target platform
- Conceptually similar to threads
  - Owner of control flow
  - Local memory

Processes (cont'd)

From an implementation standpoint, processes differ from threads in the following ways:

- Unshared heap memory may be explicitly declared for a single process, but global variables are generally not supported
- Processes are assigned to an independently synchronized processor or block of logic
- Communication and synchronization occur with hardware buffers (no OS support)
- Processes must be defined at runtime

An Impulse C Process



Process Example

```
void img_proc(co_stream pixels_in, co_stream pixels_out) {
    int nPixel;
    ...
    do {
        co_stream_open(pixels_in, O_RDONLY, INT_TYPE(32));
        co_stream_open(pixels_out, O_WRONLY, INT_TYPE(32));
        while ( co_stream_read(pixels_in, &nPixel, sizeof(int)) == 0 ) {
            ...
            // Do a filtering operation here using standard C
            ...
            co_stream_write(pixels_out, &nPixel, sizeof(int));
        }
        co_stream_close(pixels_in);
        co_stream_close(pixels_out);
    } while(1); // Run forever...
}
```

Process Creation

```
#define BUFSIZE 4
void my_app_configuration() {
    co_process procHost1, procPe1, procPe2;
    co_stream s1, s2;
    s1 = co_stream_create(‘‘s1’’, INT_TYPE(16), BUFSIZE);
    s2 = co_stream_create(‘‘s2’’, INT_TYPE(16), BUFSIZE);
    procHost1 = co_process_create(‘‘Host1’’, (co_function)Host1
        , 1, s1);
    procPe1 = co_process_create (‘‘Pe1’’, (co_function)Pe1, 2,
        s1, s2);
    procPe2 = co_process_create (‘‘Pe2’’, (co_function)Pe2, 1,
        s2);
}
```

Process Creation (cont'd)

`co_process_create`

- This function is used to define both hard are and software processes (software unless otherwise specified)
- Three arguments:
  1. Pointer to character string (NULL terminated) that contains process name
  2. Function pointer of type `co_function`, which identifies the specific run function that is to be associated with the call to `co_process_create`
  3. Number of input and output ports that are connected to the process, with a list of actual ports (i.e. streams, signals) that follow

Communication Interfaces

- `co_stream`
  - A streaming point-to-point interface on which data is transferred via a FIFO buffer interface
- `co_signal`
  - A buffered point-to-point interface on which messages may be posted by sending a process and waited for by a receiving process
- `co_memory`
  - A shared memory interface supporting block reads and writes. Memory characteristics are specific to the target platform.
- `co_register`
  - A low-level, unbuffered hardware interface.

Streams

- Most common communication interface between Impulse C processes
- Unidirectional communication channels that connect multiple processors, whether hardware or software
  - Implemented in hardware as FIFO buffers
- In a dataflow-oriented Impulse C application streams are read from and written to as data becomes available
  - If no data is available on the stream being read, the process blocks until such time as data is made available by the upstream process
- Choose the buffer size carefully, as the width and depth of a stream will have a significant impact on the amount of hardware required to implement the process

Input Streams

- Two operations may be performed on an input stream
  - `co_stream_eos` - End-of-stream test checks to see whether a “close” operation was performed on the stream by the upstream process
  - `co_stream_read` - Attempts to read the next stream element and blocks if the stream is empty
- The method of reading from a stream depends on the nature of your application
- Efficient use of stream reads (preferred method)

```
err = co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));
while (co_stream_read(input_stream, &value, INT_TYPE(32))==
          co_err_none) {
     // Process value here...
}
co_stream_close(input_stream);
```

Output Streams

- Output streams may be written to using the `co_stream_write` function

```
co_stream_open(output_stream, O_WRONLY, INT_TYPE(32));
for (I = 0; I < ARRAYSIZE; i++) {
    co_stream_write(output_stream, &data[i], sizeof(int32)
        );
}
co_stream_close(output_stream);
```

- The stream must be a writable stream, which has been opened with the `O_WRONLY` direction indicator, and the data must match the size of the stream datatype
- `co_stream_write` first checks to see if the specified output stream is full and blocks until there is room to place the data in the stream

Stream Creation

- `co_stream_create` creates a stream, defines its data width and its buffer size, and makes the stream available for use in subsequent `co_process_create` calls
  1. Optional name that may be assigned to the stream for debugging, external monitoring, and post-processing purposes
  2. The type and size of the system's data element. Macros are provided for defining specific types (INT_TYPE, UINT_TYPE, CHAR_TYPE)
  3. Buffer size, which directly relates to the size of the FIFO buffer that will be created between the two processes that are connected with the stream

```
#define BUFSIZE 4
co_stream stream = co_stream_create("stream", INT_TYPE(32)
    , BUFSIZE);
```

Deadlocks

- A stream deadlock occurs when one process is unable to proceed with its operation until another process has completed its tasks and written data to its outputs
  - If the two processes are mutually dependent or are dependent on some other blocked process, the system can quickly come to a halt
- Most deadlock problems can be fixed by increasing the stream depth
- Another common solutions is to use nonblocking stream reads (`co_stream_read_nb`)
- The programmer has to be aware of the dependencies between processes and their use of the streams in order to avoid deadlocks in the first place

Signals

- Signals allow the programmer to gain more direct control over the starting, stopping, and synchronization of processes
- Signals allow processes to communicate using a message passing scheme
- Read (or wait) operations are blocking, while write operations are non-blocking
  - `co_stream_post` - post a message to the receiving process
  - `co_stream_wait` - wait until a message has been sent by the sending process
- The most ideal form of synchronization between processes (e.g. handshaking)

Signal Usage

Producer

```c
void proc1_run(co_signal ackSignal, ...) {
    ...
    co_signal_post(ackSignal, 1); // post go-ahead signal to other process
    ...
}
```

Consumer

```c
void proc2_run(co_signal ackSignal, ...) {
    ...
    co_uint32 trigger;
    co_signal_wait(ackSignal, &trigger); // wait for the go-ahead
    // now proceed as usual
    ...
}
```

Shared Memory

- An alternative to stream-based communication
- Can be useful for initializing a process with some frequently used array values
- May be a more efficient, higher-performance means of transferring data between hardware and software processes for some platforms
- Careful synchronization is required when using shared memory as communication line, usually through the use of signals

Shared Memory Configuration

- Unlike streams, they require a platform-specific identifier that indicates the physical location of the memory resource
  - This location can be found in the XML files in `/Impulse/Codeveloper3/Architectures`
- `co_memory_create` - allocates a specified number of bytes of memory for reading and writing and returns a handle that can be used to access the memory
  1. Optional name that may be assigned to the memory for debugging, external monitoring, and post-processing purposes
  2. String indicating physical location for memory on platform
  3. Number of bytes to allocate for memory

```
void config_hello(void *arg) {
    co_memory memory;
    memory = co_memory_create(‘‘Memory’’, ‘‘mem0’’, MAXLEN*
        sizeof(char));
    ...
}
```

Shared Memory Usage

Producer

```
void Producer(co_memory shared_mem, ...) {
    static char HelloWorldString[] = ÒHello FPGA!Ó;
    co_uint32 count = strlen(HelloWorldString);
    co_memory_writeblock(shared_mem, 0, HelloWorldString, count
        );
    ...
}
```

Consumer

```
void DoHello(co_memory shared_mem, ...) {
    char buf[MAXLENGTH];
    co_uint32 count = MAXLENGTH;
    co_memory_readblock(shared_mem, 0, buf, count);
    ...
}
```