An introduction to the C7000 Compiler

April 2023

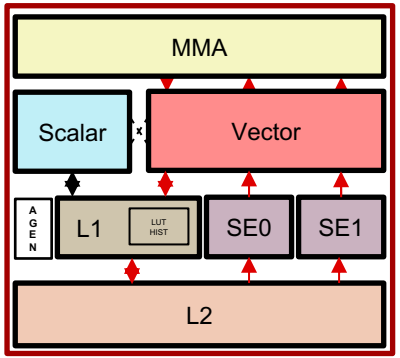
Topics

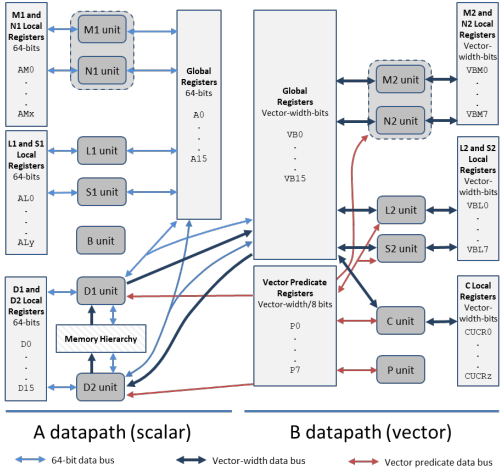
- Introduction to C7000
  - C7000 CPU overview
  - C7000 toolchains
- Basic C7000 programming
  - Documentation and usage
  - Intrinsics and header files
  - Vector programming
- Advanced C7000 programming
  - SE and SA usage
  - Predicated loads and stores
- Optimizing code for C7000
  - Software pipelining
  - Interpreting software pipelining feedback
- C6x migration

C7000 compiler toolchains, diagram, and variants

INTRODUCTION TO C7000

C7000 CPU Diagram





- Vector machine (SIMD)
  - C7100 vector registers are 512 bits
    - Vector `int` ADD is 16x additions
- Lots of functional units (VLIW)
  - 13 units + some “no unit” instructions
  - Up to 12 instructions in parallel
- Data forwarding and addressing engines
  - 2x Streaming Engines (SE) provide additional load bandwidth (1 vector width each per cycle)
  - 4x Streaming Address Generators
- Matrix Multiply Accelerator (MMA)
- For best performance:
  - Maximize vector usage (SIMD)
  - Maximize instruction parallelism (VLIW)
  - **The C7000 compiler helps on both fronts**

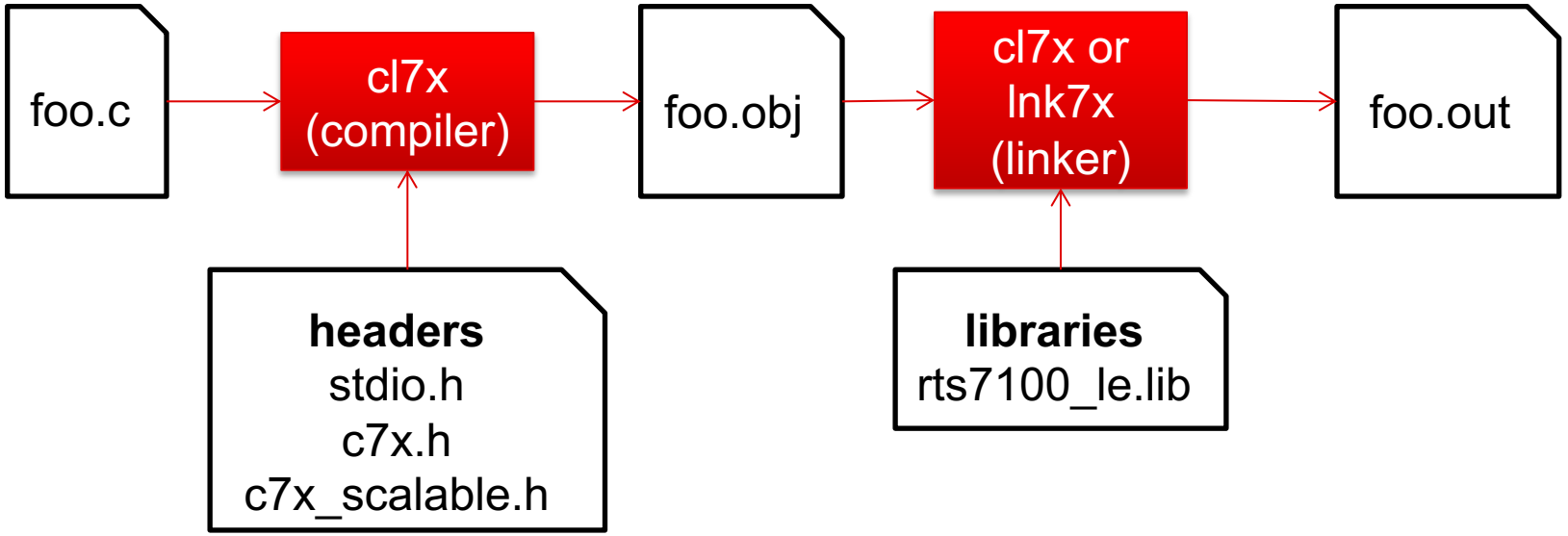
C7000 C/C++ Optimizing Compiler Tools

- Compiler enables full-access to all C7000+MMA ISA features from C/C++
  - Version 3.1.LTS available now (https://www.ti.com/tool/C7000-CGT)
  - Performance entitlement available from C/C++. No need to code in assembly.
- Host Emulation package allows initial development, execution, and debug on the host
- Scalable vector programming model allows writing vector code that is portable across variants of C7000 with varying vector widths
- Vector types, vector operators, and intrinsics allows C/C++ code to take advantage of C7000 vector instructions
- Automatic compiler transformations to use vector instructions from some scalar code (SIMD)

C7000 Toolchains

- Compiler (cl7x)
  - Used for generating C7000 code
  - Ships with standard C/C++ libraries
  - Ships with additional C7000 specific headers
- Host Emulation
  - Used for generating host code (x86-64) that behaves like C7000
  - Uses host standard C/C++ libraries
  - Ships with additional libraries and C7000 specific headers to behave like C7000
  - Useful for developing and debugging
- Both toolchains are in the C7000-CGT installer
  - (Where CGT is Code Generation Tools)

C7000 Compiler Toolchain (Simplified)

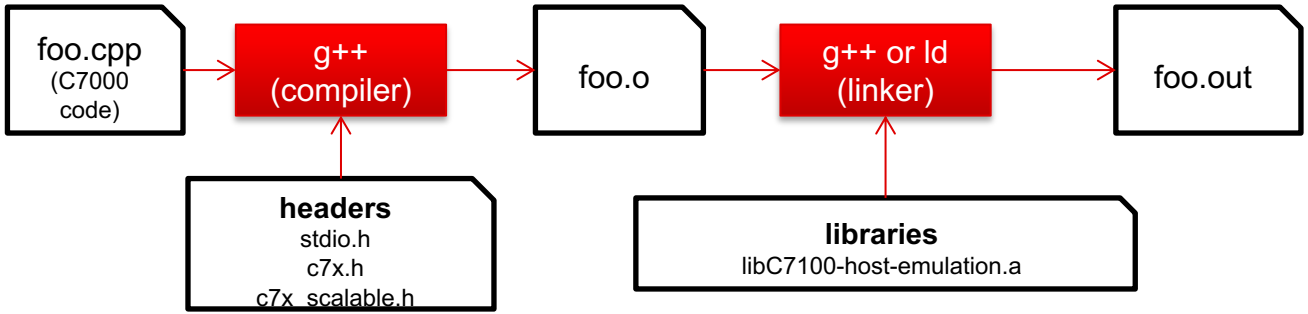



```
cl7x foo.c -z -lrts7100_le.lib -o foo.out
```

^ Invokes linker

Host Emulation

- Improves productivity by allowing users to develop/debug C7000 code on a host machine (x86-64)
  - Replaces cl7x compiler with g++ or MSVC
  - Intrinsics are provided through host (Linux/Windows x86-64) libraries
- Once initial development is complete, the same code can be compiled unmodified with the cl7x compiler
- Host Emulation User's Guide: http://www.ti.com/lit/pdf/spruig6




```
g++ --std=c++14 -fno-strict-aliasing foo.c -I$HOSTEM_DIR/include/C7100 -L$HOSTEM_DIR -lC7100-
host-emulation -o foo.out
```

Documentation, compiler options, vectors, and intrinsics

BASIC C7000 PROGRAMMING

Compiler Documentation

- C7000 C/C++ Optimizing Compiler User's Guide (SPRUIG8)
  - https://www.ti.com/lit/pdf/spruig8
- C7000 Optimization Guide (SPRUIV4)
  - https://www.ti.com/lit/pdf/spruiv4
- C7000 Host Emulation Guide
  - http://www.ti.com/lit/pdf/spruig6
- C6000 to C7000 Migration User's Guide
  - https://www.ti.com/lit/pdf/spruig5
- Learning Software Pipeline Feedback Information
  https://www.youtube.com/watch?v=QxNZbCny-hU

Compiler Options

- -mv7100 / -mv7120 / -mv7504: Choose the C7000 variant
- -o2 / -o3: High level of optimization (Recommended)
- -o4: Link-time optimization
  - Increases compile time considerably
- -k: Keeps assembly (.asm) file.
  - Contains software pipeline information feedback
- -mw: Adds more information to software pipeline information feedback in .asm file
- -os: Prints optimizer comments in the .asm file.
  - Helpful for determining what the optimizer did to your code.
- -mo: Places functions in own subsections so linker can remove unreferenced functions.
  - Tradeoff: Increases code size because of required padding for each function/section.
- Unneeded: -g
  - Debug is always on. Becomes less useful as -o<n> gets higher. Consider -o=off or -o0 if you need to debug.
- --symdebug:none: Turns off debug info
  - Reduces debug directive clutter in .asm

Headers and APIs

- Programming interface to certain HW features defined (in-part) with intrinsics: `c7x*.h` files in `include` directory
- Primary headers
  - c7x.h – Contains high level intrinsics that are descriptively named (Section 5.15.1)
    - Includes c7x_direct.h – Contains alternative intrinsics that are named like instructions. (Section 5.15.3)
    - Includes c7x_vpred.h – Contains intrinsics that use __vpred instead of boolean vectors. (Note: Boolean vectors are recommended.)
    - Includes c7x_strm.h – Contains Streaming Engine and Streaming Address Generator intrinsics, enums, macros, and structs. (Section 4.14)
    - Includes c7x_mma.h – Contains Matrix Multiply Accelerator intrinsics, enums, macros, and structs.
    - Includes c7x_luthist.h – Contains lookup table and histogram intrinsics and macros.
  - c7x_scalable.h – Contains **C++** utilities for scalable vector programming of C7000.
  - c6x_migration.h – Contains implementations of legacy C6000 intrinsics. (Section 5.15.6)

Intrinsics

- High level intrinsics
  - Overloaded, better for portability
  - Located in include/c7x.h
  - Some complicated instructions are not accessible via “generic” intrinsics

- “Direct” Intrinsics
  - Used when programmer needs exact instruction
  - Useful when operand interleaving or result deinterleaving is required
  - Located in include/c7x_direct.h
  - Some complicated instructions are only accessible with “direct” intrinsics

Scalable Vector Programming

- C7000 series DSPs may have different vector widths. E.g., C7504 is 256 bit, C7100 is 512 bit.
- C7000 APIs are written in terms of specific vector types that fit on the target.
  - int8, int16, etc.
- If an algorithm is written in terms of a specific vector width, it by definition doesn't scale between targets.
  - int8 for this target, int16 for that target…
- The c7x_scalable.h header provides target dependent types and related utilities to abstract out the vector width.
  - E.g., int_vec is a vector of int. The number of elements depends on the target.

Scalable Vector Programming

- Enables users to write code that is portable across C7000 variants (E.g., 256/512-bit vector widths)
- Include the c7x_scalable.h header
- The header provides vector properties (1) as well as common typedefs.
  - `int_vec` is an `int16` on 512-bit targets and an `int8` on 256-bit targets.
  - `element_count_of<int_vec>::value` is `16` on 512-bit targets and `8` on 256-bit targets.
  - For example:

```
char_vec *in = (char_vec*)my_char_array;
long sum = 0;
for (int i = 0; i < length / element_count_of<char_vec>::value; i++) {
    sum += __horizontal_add(in[i]);
}
```

- (1) C++ is designed to be able to query the properties of a type at compile time. These APIs are often referred to as "type traits"
- Refer to the C7000 C/C++ Compiler Users Guide, section C7000 Scalable Vector Programming for details

Scalable Vector Programming – Example

```
#incude <c7x_scalable.h>

using namespace c7x;

void vector_sum(int_vec *in1, int_vec *in2, int_vec *out, int n)
{
    // Assumes n is multiple of vector length

    int_vec *vec_in1 = (int_vec*) in1;
    int_vec *vec_in2 = (int_vec*) in2;
    int_vec *vec_out = (int_vec*) out;
    int i;

    for (i = 0; i < n/element_count_of<int_vec>::value; i++)
    {
        vec_out[i] = vec_in1[i] + vec_in2[i];
    }
}
```

- Notes:
  - Considering the vector length has been factored out by:
    - Changing types
    - Changing “magic” constants that represented the vector length
- (This example will not generate efficient code *yet* due to a recurrence bound.)

Vector Programming – Basics

- Vector types are an extension to the C/C++ languages
  - Similar to the OpenCL language (v1.2 OpenCL standard)
  - Types written as <base_type><num_elem>, e.g. **char64**
- Vector operators: +, -, *, etc.
  - Generally, requires exact types. (No implicit conversions.)
- Intrinsics accept and return vectors – See c7x.h and related headers.
  - Includes the Streaming Engine, Streaming Address Generator, and Matrix Multiply Accelerator.
- Vector accessors
  - Can be used as left-hand side values. (E.g. vec.s0() = 1;)
  - .sn() where n is 0-f (E.g. vec.s3() or vec.sf())
    - Multiple element “swizzles” are available in cl7x, but not in host emulation. (E.g. vec.s1254())
  - .r(), .i() – Real and imaginary component of complex types.
  - .even(), odd(), .lo(), .hi()
  - .s[n] where n is an integral value (E.g. vec.s[3] or vec.s[i])
    - Equivalent to .sn() when n is an immediate. Equivalent to an array access when it is not.
- Vector initializers
  - int16(0) – All elements
  - int8(0,1,2,3,4,5,6,7) – Individual elements
  - int8(int4_vec_a, int4_vec_b) – Can also be composed from vectors.

Vector Programming – Example

```
void vector_sum(int *in1, int *in2, int *out, int n)
{
    // Assumes n is multiple of 16

    int16 *vec_in1 = (int16*) in1;
    int16 *vec_in2 = (int16*) in2;
    int16 *vec_out = (int16*) out;
    int i;

    for (i = 0; i < n/16; i++)
    {
        vec_out[i] = vec_in1[i] + vec_in2[i];
    }
}
```

- Notes:
  - In typical use cases, vectors are being loaded/stored from memory.
  - The result type of `vec_in1[i]` is `int16`.
  - An array access on a vector pointer scales by the element count. (16 `int`s in this example.)
  - The “+” is a vector addition.
  - Assuming/requiring data lengths in multiples is not ideal.

More later!

- (This example will not generate efficient code *yet* due to a recurrence bound.)

More later!

C++ Guidelines

- Compiler supports C++14
- These features have potential run-time overheads. Consider whether the benefits are worth the cost:
  - Calls to new(), although this is essentially no more or less expensive than malloc()
  - Use of the Standard Template Library (STL), mainly due to hidden calls to new()
  - Exceptions / exception handling
  - Run-Time Type Information (RTTI)
  - Multiple inheritance
  - Virtual functions (although the run-time cost is usually small)
- Use these features freely, as they have little to no run-time overhead:
  - Templates
  - Operator overloading
  - Function overloading
  - Inlining
  - Well-designed inheritance. Calling a member function of a derived class incurs no penalty if the object type is known at compile-time.
- The following features may/often improve performance and should be used where possible:
  - Use of const
  - Use of constexpr
  - Passing objects by-reference instead of passing objects by-value
  - Constructs and expressions that can be evaluated at compile-time versus run-time

Streaming engine, streaming address generator, vector predication,

and more scaling

ADVANCED C7000 PROGRAMMING

Streaming Engine (SE)

- The streaming engines provides a flexible, high-bandwidth mechanism for reading large quantities of data into the DSP CPU via two independent, programmable streams
- Benefits:
  - Prefetches data from above L2 cache
  - Allows user to count and offset for each dimension of data, up to 6 dimensions
  - Computation of addresses is done automatically – no need for compiler to generate address calculation instructions that can slow the loop. May lead to better loop transformations in the compiler.
  - Typically allows user to “flatten” loop nests, allowing compiler to pipeline more code.
  - Several data formatting and pattern access features
- Two Streaming Engines, SE0 and SE1

- Compiler does not automatically use SE. Programmer must program them with C/C++ interface provided.
- See include/c7x_strm.h for example usage

SE conceptual operation

```
// This defines the order in which the bytes are presented to the stream data FIFO.
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++) {
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++) {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++) {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++) {
                ptr2 = ptr; // save current position before entering next level
                for (i1 = 0; i1 < ICNT1; i1++){
                    ptr1 = ptr; // save current position before entering next level
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        fetch_and_place_in_FIFO( ptr, ELEM_BYTES);
                        ptr = ptr + ELEM_BYTES;
                    }
                    ptr = ptr1 + DIM1;
                }
                ptr = ptr2 + DIM2;
            }
            ptr = ptr3 + DIM3;
        }
        ptr = ptr4 + DIM4;
    }
    ptr = ptr5 + DIM5;
}
```

Note: other “modes” operate slightly differently

SE usage

Setup/Open:

```
#include <c7x.h>

__SE_TEMPLATE_v1 params = __gen_SE_TEMPLATE_v1();

params.ICNT0 = 4;
params.ICNT1 = 2;
params.DIM1  = 4;
params.ICNT2 = 2;
params.DIM2  = 8;
params.ICNT3 = 4;
params.DIM3  = -16;

params.DIMFMT    = __SE_DIMFMT_4D;
params.VECLEN    = __SE_VECLEN_4ELEMS;
params.ELETYPE   = __SE_ELETYPE_32BIT;

// OPEN STREAMING ENGINE 0 AT startaddr
// WITH PARAMETER TEMPLATE params
__SE0_OPEN((void*) startaddr, params);
```

After setup, use the SE as follows:

- `SE0(uint4)     // Does not advance SE0`
- `SE0ADV(uint4) // Advances SE0`

```
// READ THE STREAM AND ADVANCE THE COUNTERS
for (I0 = 0; I0 < 8; I0++)
{
    uint8 vdata;
    vdata.lo() = __SE0ADV(uint4);
    vdata.hi() = __SE0ADV(uint4);
    vresult += vdata;
}
// CLOSE THE STREAM
__SE0_CLOSE();
```

SA setup is similar. See c7x_strm.h

Streaming Address Generator (SA)

- Generates address offsets
- HW detail: Often used in memory operands of assembly instructions to compute a memory address. Often paired with a base address in a register. E.g. VLD4W ***D0[SA1++]**,VB1
- Multi-dimensional capability, just like Streaming Engine
- Four SA generators: SA0, SA1, SA2, SA3
- Used for stores *or* when you run out of Streaming Engines
- Compiler does not automatically use SA; User must program

C/C++ usage:

- Very similar to SE, especially in setup/open/close

```
my_uint4_var = *__SA1ADV(uint4, startaddr);
```

- See include/c7x_strm.h for examples

	Streaming Engine (SE)	Streaming Address Generator (SA)
What it "returns"	Scalar or vector data	An address + offset

Vector Predication

- Instruction predicate: controls execution of an instruction
  - E.g. `[A0] MPY` – like C6x
- Vector predicate: controls execution for part of a vector instruction
  - E.g. `VSEL P0, VB8, 0, VB13`
  - Represented in C as __vpred (low level) or __bool[n] (high level)
    - __vpred: byte basis
    - __bool[n]: element basis. Compiler "lowers" __bool[n] to __vpred
- Vector predicate registers: `P0-P7`
  - Generally, each bit of a vector predicate register controls a byte of a vector, with a total of 64 bits on C7100.
    - May be instruction dependent (additional scaling may be needed – expand/reduce intrinsics)
    - When using Boolean vectors and associated intrinsics, scaling of the predicate value occurs automatically
    - SA generates vector predicates: `PSA0-PSA3`
- Examples
  - VSEL uses the predicate register to select bytes from src1 or src2
  - Vector compares (`VCMPEQB`) write results to predicate registers
  - Predicated stores (`VSTP16W`)
  - Predicate only operations available: AND, ANDN, OR, XOR, NOR, BITR, etc.

Predicated Loads & Stores

- Why: programmer has row of data that isn't multiple of machine's vector size
  - E.g. 15 elements to store, but want to store them out 8 at a time; 1 extra element
  - Can use predicate registers or SA configuration to prevent last element from being loaded/stored.
- Explicit vs implicit (assembly, hardware detail)
  - Explicit: The operation takes a vector predicate register operand
    - Predicate may be generated from an SA or a predicate-generating instruction
  - Implicit: Predication occurs if the memory operand is using a SA
    - **More efficient**: extraction of predicate from SA occurs w/o an instruction
  - Compiler will try to optimize explicit predication to implicit
- Target support:
  - C7100: Predicated stores (implicit & explicit)
  - C7120/C75xx: Predicated stores (implicit & explicit) and implicitly predicated loads

Predicated Stores

- Normal stores in C:
  - No predication: `*ptr = int16_vector;`
    - Generates: VST16W
  - With predication: `__vstore_pred(vp, ptr, int16_vector);`
    - Generates: VSTP16W (explicit) or VST16W (implicit)

Obtaining a vector predicate from an SA

```
vpred vp = __SA0_VPRED(type);
```

- Specialized stores in C:
  - No predication: `__vstore_packl(ptr, int16_vector);`
    - Generates: VSTHSVPACKL
  - With predication: `__vstore_pred_packl(vp, ptr, int16_vector);`
    - Generates: VSTPHSVPACKL (explicit) or VSTHSVPACKL (implicit)
- May also be generated by compiler during vectorization

Predicated Loads & Stores: Pitfalls

- There are no explicitly predicated load instructions on HW
  - If an SA is not used to derive the vector predicate or the compiler is not able to convert to the implicit form, the intrinsic will become a longer sequence of instructions as appropriate.
- Efficiency: Implicit predication is tied to SA
  - Must use __SAn to retrieve pointer and __SAn_VPRED or __SAn_BOOL to retrieve predicate. Type information must agree.
- Correctness: Order of evaluation for arguments in C is not defined
  - `foo(i, i++);` When does the increment occur?
  - Similar: `__vstore_pred(__SA0_VPRED(int16), __SA0ADV);`
- Correctness:
  - If the SA is used in an unpredicated load/store (*ptr) and the SA is configured for predication, predication may occur. (Unexpected behavior)

Scalable Vector Programming: Example Memcpy

```
#include <c7x_scalable.h>
using namespace c7x;
void memcpy_scalable_strm(const char *restrict in, char *restrict out, int len) {
    int veclen = element_count_of<char_vec>::value;
    int cnt = len /veclen;
    cnt += (len % veclen > 0);

    __SE_TEMPLATE_v1 in_tmplt = __gen_SE_TEMPLATE_v1();
    __SA_TEMPLATE_v1 out_tmplt = __gen_SA_TEMPLATE_v1();
    in_tmplt.VECLEN = se_veclen<char_vec>::value;
    in_tmplt.ELETYPE = se_eletype<char_vec>::value;
    in_tmplt.ICNT0 = len;
    out_tmplt.VECLEN = sa_veclen<char_vec>::value;
    out_tmplt.ICNT0 = len;
    __SE0_OPEN(in, in_tmplt);
    __SA0_OPEN(out_tmplt);

    for (int i = 0; i < cnt; i++) {
        char_vec tmp = strm_eng<0, char_vec>::get_adv();
        __vpred pred = strm_agen<0, char_vec>::get_vpred();
        char_vec_ptr addr = strm_agen<0, char_vec>::get_adv(out);
        __vstore_pred(pred, addr, tmp);
    }
    __SE0_CLOSE();
    __SA0_CLOSE();
}
```

Goal: fast memcpy, using SE and SA without considering the target width.

Considerations:

1. Vector width is target dependent.
2. SE/SA encode specific vector lengths.
3. To store a partial vector, SA predication is needed.

Get the number of bytes in *a vector*.

Configure SE/SA fields for *a vector*.

Read in *a vector* of data from SE

Get a predicate and *a vector* address for the store from SA

Store up to *1 vector*.

Scalable Vector Programming: Example Memcpy

```
#include <c7x_scalable.h>
using namespace c7x;
void memcpy_scalable_strm(const char *restrict in, char *restric
    int veclen = element_count_of<char_vec>::value;
    int cnt = len /veclen;
    cnt += (len % veclen > 0);

    __SE_TEMPLATE_v1 in_tmplt = __gen_SE_TEMPLATE_v1();
    __SA_TEMPLATE_v1 out_tmplt = __gen_SA_TEMPLATE_v1();
    in_tmplt.VECLEN = se_veclen<char_vec>::value;
    in_tmplt.ELETYPE = se_eletype<char_vec>::value;
    in_tmplt.ICNT0 = len;
    out_tmplt.VECLEN = sa_veclen<char_vec>::value;
    out_tmplt.ICNT0 = len;
    __SE0_OPEN(in, in_tmplt);
    __SA0_OPEN(out_tmplt);

    for (int i = 0; i < cnt; i++) {
        char_vec tmp = strm_eng<0, char_vec>::get_adv();
        __vpred pred = strm_agen<0, char_vec>::get_vpred();
        char_vec_ptr addr = strm_agen<0, char_vec>::get_adv(out)
        __vstore_pred(pred, addr, tmp);
    }
    __SE0_CLOSE();
    __SA0_CLOSE();
}
```

cl7x -mv7100 –o3 yields:

```
    [ A1]   ADDW    .L1     A1,0xffffffff,A1
||  [ A0]   ADDW    .D1     A0,0xffffffff,A0
||          BNL     .B1     ||$C$L2||
||  [!A0]   VST64B  .D2     VB0,*D0[SA0++]
||  [!A1]   VMV     .L2     SE0++,VB0
||          TICK
```

cl7x -mv7504 –o3 yields:

```
    [ A1]   ADDW    .L1     A1,0xffffffff,A1
||  [ A0]   ADDW    .D1     A0,0xffffffff,A0
||          BNL     .B1     ||$C$L2||
||  [!A0]   VST32B  .D2     VB0,*D0[SA0++]
||  [!A1]   VMV     .L2     SE0++,VB0
||          TICK
```

Boolean vectors vs __vpred

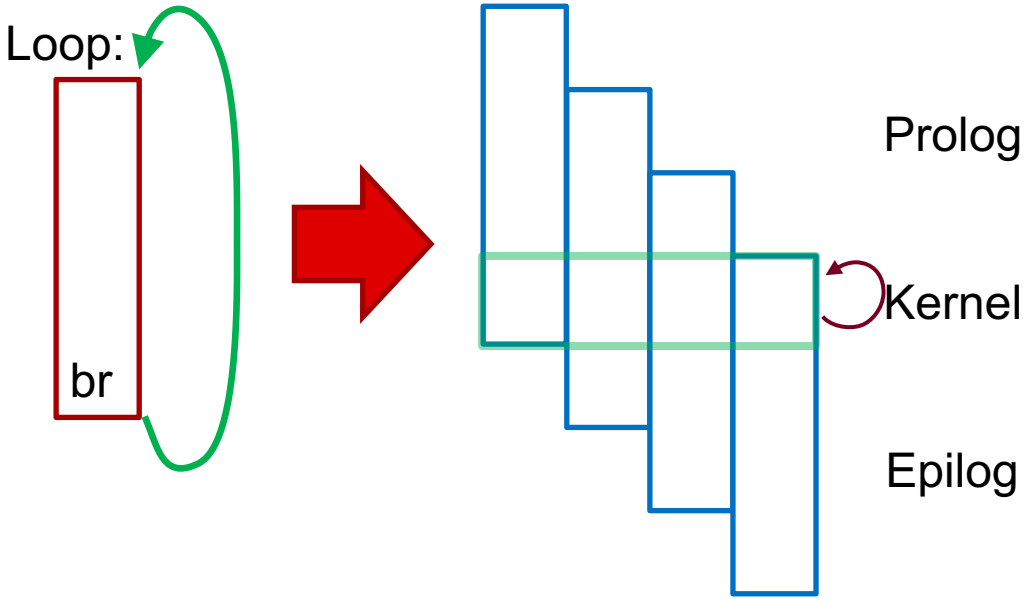
- Boolean vector types (E.g. bool16) provide a view of vector predicates (__vpred) that is consistent with other vectors.
- Boolean vectors will be lowered and optimized as __vpreds by the C7000 compiler v3.0.
  - The compiler handles considerations such as bits predicate per element in operations. (Scaling)
  - Provides a friendlier and consistent interface for initializing and modifying elements when compared to __vpred.
  - In cases where every extra cycle matters, __vpreds can still be used for greater control.
  - Direct intrinsics (low level, assembly like) always use __vpreds.

Software pipelines, restrict, and pragmas

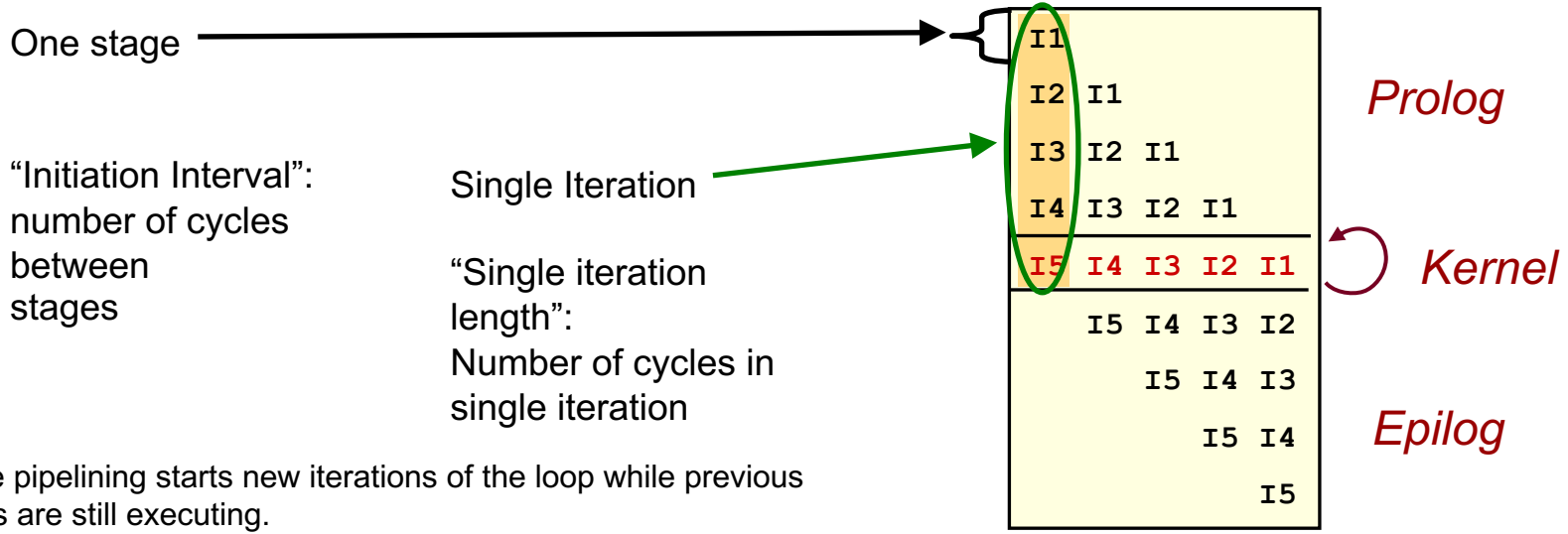
OPTIMIZING FOR C7000

Software Pipelining

- VLIW architectures require optimization called *software pipelining* for performance
- Overlap successive iterations of loop
  - Multiple iterations of loop iterating in parallel
  - Start successive iteration while previous iteration is still executing
- Multiple, pipelined functional units allow several instructions to execute at the same time
- Compiler automatically performs software pipelining
  - Compiler partitions instructions to scalar (A) or vector (B) side
  - Compiler utilizes the various register files and functional units



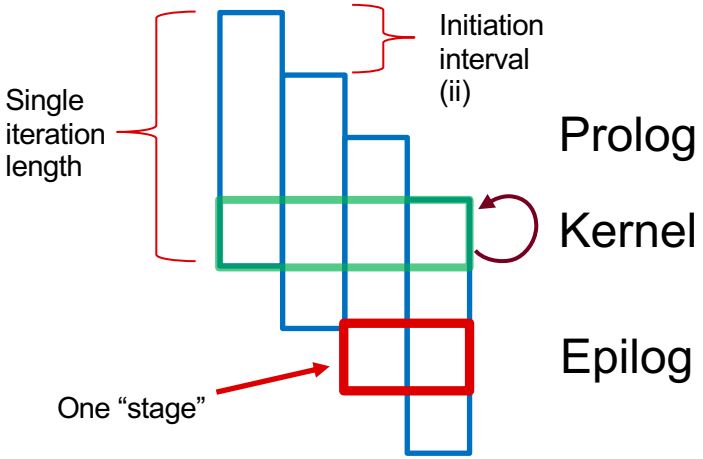
Software Pipelining Visualized



- Software pipelining starts new iterations of the loop while previous iterations are still executing.
- In some cases, effectiveness limited by:
  - Legality: iteration-to-iteration memory dependences and data dependences
  - Number of functional units and registers
- Software pipelined loops on C7000 are interruptible

Software pipeline terminology (1)

- Initiation interval
  - AKA iteration interval
  - Number of cycles between start of each iteration
  - Number of cycles in one stage
- Single iteration length
- Stage
  - ii cycles long
  - Stage of the prolog/epilog
  - Stage of a single iteration
  - Kernel is always ii cycles long
  - Most stages are ii cycles long
    - 1st prolog, last epilog may be shorter
- Minimum safe iteration (“trip”) count
  - Number of iterations the loop must execute in order for the software pipelined loop to be used safely
- Number of overlapped iterations
  - (Four overlapped iterations in the SWP loop)



Software pipeline terminology (2)

- Single assignment:
  - Doesn't require registers to be read that have pending results

- Multiple assignment:
  - A write may be in flight to a register while that same register's value is being read
  - Multiple in-flight writes to a register

```
; Example on C6000
ADD.S1  A7, A8, A0
LDW.D1  *A0, A1       <- Load takes 5 cycles (4 cycle delay)
MPY.M1   A1, A2, A3   <- Uses current value of A1
NOP 3
SHR.S1  A3, 15, A3    <- A1 now holds the result of the LDW
ADD.L1  A3, A4, A4
```

Getting the compiler to software pipeline a loop

- Use –o2 or higher optimization.
- Use –mw for more detailed software pipeline feedback in the assembly file (.asm). Use -k to keep .asm
- Loop must not contain function calls
  - Use inline pragmas/attributes applied to functions/function calls.
  - See section 3.10 in the C7000 C/C++ Compiler User's Guide

- Compiler must be able to remove all control-flow within loop body
  - Compiler will attempt to "if-convert" code inside of loop body
    - Complex, nested if-statements may not be if-converted
  - Loops with "Early-exit" **`break`** statements or goto's often cannot be pipelined
- Best practices for obtaining best performance:
  - Use signed loop counters, loop iteration variables.
    - Unsigned "wrap-around" is defined behavior in C/C++. Some optimizations won't occur
  - Use "restrict" on input and output pointers *when legal*
  - Use MUST_ITERATE pragma to indicate min, max, and multiple for loop's possible iteration count
- See section 4.7 and 4.8 in the C7000 C/C++ Compiler User's Guide

Software pipeline feedback (1)

```
;*----------------------------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file              : F_cfir_c.c
;*      Loop source line                : 43
;*      Loop closing brace source line  : 52
;*      Known Minimum Trip Count        : 4
;*      Known Max Trip Count Factor     : 1
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound    : 10
;*      Partitioned Resource Bound      : 10 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 10 Did not find schedule
;*         ii = 11 Register is live too long
;*         ii = 11 Did not find schedule
;*         ii = 12 Did not find schedule
;*         ii = 13 Register is live too long
;*         ii = 13 Schedule found with 3 iterations in parallel
```

Software pipeline feedback is placed in the assembly (.asm) file.

- **Loop found in file, Loop source line, Loop opening brace source line, Loop closing brace source line**
- **Known Minimum/Maximum Trip (Iteration) Count**

- **Initiation interval (ii):** In the example, the compiler was able to construct a software pipelined loop that starts a new iteration every 13 cycles. The smaller the initiation interval, the fewer cycles it will take to execute the loop.
- **Iterations in parallel:** When in the steady-state (kernel), the example loop is executing different parts of three iterations at the same time. This means that before iteration n has completed, iterations n+1 and n+2 have begun.

Software pipeline feedback (2)

```
;*----------------------------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file               : F_cfir_c.c
;*      Loop source line                 : 43
;*      Loop closing brace source line   : 52
;*      Known Minimum Trip Count         : 4
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 10
;*      Partitioned Resource Bound       : 10 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 10 Did not find schedule
;*         ii = 11 Register is live too long
;*         ii = 11 Did not find schedule
;*         ii = 12 Did not find schedule
;*         ii = 13 Register is live too long
;*         ii = 13 Schedule found with 3 iterations in parallel
```

Software pipeline feedback is placed in the assembly (.asm) file.

- **Loop-Carried Dependency Bound**
  - The distance of the largest loop carry path, if one exists. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration
- **Unpartitioned Resource Bound**
  - The best case resource bound minimum initiation interval (mii) before the compiler has partitioned each instruction to the A or B side.
- **Partitioned Resource Bound (pre-sched, post-sched)**
  - The mii after instructions are partitioned to the A and B sides. Pre-scheduling and post-scheduling values are given. The post-scheduling value is the partitioned resource bound after scheduling occurs. Scheduling sometimes involves the addition of instructions, which may affect the resource bound.

Software pipeline feedback (3)

```
;*        SINGLE SCHEDULED ITERATION
;*
;*        ||$C$C234||:
;*   0              TICK                                ; [A_U]
;*   1              GETP    .L1     0,0,A1 ; pre        ; [A_L1]
;*     ||           GETP    .S1     0,0,A2 ; post       ; [A_S1] |47|
;*   2              MVDLY3  .S1     A1,A0               ; [A_S1] Split a
long life
;*     ||   [ A1]   ADDAW   .D1     D1,D0,A3            ; [A_D1] CASE-1
;*     ||   [ A1]   MV      .M1     AM1,A4              ; [A_M1] CASE-1
;*     ||   [ A2]   ADDW    .D2     D0,0x8,D0           ; [A_D2] |43|
CASE-1
;*   3              LDUW    .D2     *A4(4),B3           ; [A_D2] |48|
;*     ||           LDUW    .D1     *A3(12),BM0         ; [A_D1] |48|
;*   4              LDUW    .D2     *A4(0),B4           ; [A_D2] |48|
;*     ||           LDUW    .D1     *A3(8),BM1          ; [A_D1] |48|
;*     ||           ADDD    .M1     A4,0x8,A4           ; [A_M1] |47|
;*   5      [ A0]   MV      .S1X    B12,A6              ; [A_S1]  ^
;*     ||           LDUW    .D1     *A3(0),AM0          ; [A_D1] |48|
;*     ||           LDUW    .D2     *A3(4),BM1          ; [A_D2] |48|
;*   6      [ A0]   MV      .M2     B12,BM5             ; [B_M2]  ^
;*     ||   [ A0]   MV      .M1X    B12,AL0             ; [A_M1]  ^
;*     ||           LDUW    .D1     *A3(24),BM3         ; [A_D1] |48|
;*     ||           LDUW    .D2     *A3(28),B11         ; [A_D2] |48|
(truncated)
```

- -mw will provide a “single scheduled iteration” view of the pipeline.
  - In the kernel, it may be hard to discern the data flow due to multiple assignment and overlapped iterations.
  - The “single scheduled iteration” provides a view of what a single iteration of a source loop will do without needing to consider multiple iterations in parallel in the kernel.
  - Left hand side numbers are the cycle indexes.

Loop-carried dependence bound

- If loop-carried dependence bound is > partitioned resource bound, then:
  - Code has an iteration-to-iteration memory dependence (e.g. store to load), or
  - Code has an iteration-to-iteration data dependence
  - That is, the next iteration(s) is dependent upon data from the current iteration

```
void vector_sum(int * in1, int * in2, int * out, int n)
{
    for (int i = 0; i < n; i++)
    {
        out[i] = in1[i] + in2[i];
    }
}

// cl7x -k -mv7100 -o2 -mw -os --symdebug:none vector_sum.cpp
```

```
;*----------------------------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file              : vector_sum.cpp
;*      Loop source line                : 5
;*      Loop opening brace source line  : 6
;*      Loop closing brace source line  : 8
;*      Known Minimum Trip Count        : 1
;*      Known Max Trip Count Factor     : 1
;*      Loop Carried Dependency Bound(^) : 8
;*      Unpartitioned Resource Bound    : 2
;*      Partitioned Resource Bound      : 2 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 8  Schedule found with 2 iterations in parallel
```

Adding restrict keyword

If a loop-carried dependence is caused by a memory dependence, the user may be able to use the *restrict* keyword to tell the compiler there is no pointer aliasing.

```
void vector_sum(int * restrict in1, int * restrict in2, int * restrict out, int n)
```

Only use *restrict* on a pointer declaration when the only access to those memory elements pointed to by the pointer are accessed via that restrict-qualified pointer and not another pointer.

In other words, don't access the same elements of an object via another pointer when using *restrict*.

Using restrict is a contract with the compiler!

Previous example + restrict:

```
;*      Loop Unroll Multiple              : 16x     <- Often an indication of SIMD/vectorization by the compiler
;*      Known Minimum Trip Count          : 1
;*      Known Max Trip Count Factor       : 1
;*      Loop Carried Dependency Bound(^) : 1       <- LCD Bound is now 1 (was previously 8)
;*      Unpartitioned Resource Bound      : 2
;*      Partitioned Resource Bound        : 2 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 5 iterations in parallel
```

Interpreting software pipeline feedback

- Further reading: Chapter 4 in SPRUIVA, C7000 C/C++ Optimization Guide

Attributes, Keywords, Pragmas, _nassert: helping the compiler software pipeline a loop

- Information pragmas
  - Give the compiler as much information as possible
  - Loop pragma: `#pragma MUST_ITERATE(min, max, multiple)`
  - Loop pragma: `#pragma PROB_ITERATE(min, max, multiple)`

- "I'd like to" pragmas, keywords
  - Inline keyword
  - Loop pragma: `#pragma UNROLL(unroll_amt)`
  - `#pragma FUNC_ALWAYS_INLINE` / `FORCEINLINE` (__attribute__((always_inline)) is preferable. Note: compiler supports many GCC attributes)

- Contractual pragmas
  - Loop pragma: `#pragma LOOP_NO_ALIAS` (not documented, expert use only)
- _nassert(true_expression)
  - `_nassert(my_loop_iteration_count == 10)`

Before vectorizing a loop, the compiler tries to determine if the vectorization will improve performance. It is helpful if the compiler has information about the iteration counts of the loop so the compiler can make better predictions about the profitability of vectorization. In the same way, the compiler also tries to determine if certain loop optimizations and loop-nest optimizations will be profitable and so information about the iterations counts of the loops can be helpful to the compiler.

Why are there duplicate loops in my assembly code?

- Duplicate loops are sometimes created by the compiler
  - Two assembly loops for the same C/C++ loop
  - One assembly loop will be software pipelined. The other will be not be software pipelined.
- Why are the redundant loops being generated?
  1. Minimum safe trip (iteration) count for pipelined loop not guaranteed
     - Some software pipelined loops have a minimum iteration count to be executed safely
     - Minimum safe iteration count of the software pipeline loop is not guaranteed via the compiler's analysis of the code
     - Compiler creates non-software pipelined loop to handle small iteration counts that won't be legal in the software pipelined version [We call this a "duplicate" loop. Will see comment in software pipeline information feedback.]
     - Redundant loops are not desirable because the extra control-flow (compares and branch instructions) needed to choose between the two loops.
     - See software pipeline information in .asm file
  2. Peeled loop due to vectorization
     - Loop to handle the remainder iterations after vectorization if loop counter not multiple of vectorization amount
  3. Run-time alias disambiguation
     - Compiler may generate two loops, one assuming aliasing between two pointers, one assuming no aliasing. [Rare]
- If possible, let compiler know what the range of loop iteration counts are via MUST_ITERATE pragma

MIGRATION FROM C6000

Migration from C6x code

- C6x C Code with intrinsics is supported with the C7000 compiler
  - #include <c6x_migration.h>
- Exceptions:
  - Memory-mapped registers
  - Absolute addresses
  - Control registers
- Some code using certain C6x intrinsics may not be as performant as the C6x-to-C7x instruction/intrinsic mapping is not 100% 1-to-1 and sometimes results in two or more instructions instead of one

Unnecessary/removed compiler options

- The following options that exist on the C6000 compiler do not exist on the C7000 compiler and are unnecessary:
  - -mt (telling the compiler certain function parameter aliasing does not exist)
    - We concluded this was dangerous
    - Use the restrict keyword instead (when legal)
  - -mh (controlling amount of allowable load speculation)
    - C7000 compiler automatically uses speculatable loads when necessary
  - -g (option still there, it's just unnecessary)
    - C7000 compiler automatically turns on non-intrusive debug