INF5430: High level synthesis

UNIVERSITY OF OSLO

Overview

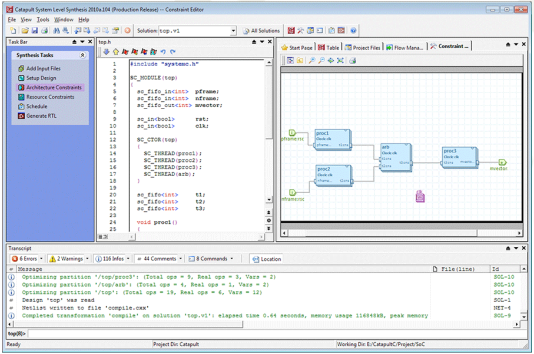
- High Level Synthesis (HLS)
- Designing hardware with C
- Compiler transformations
- Non-Optimal code for Synthesis
- References

High Level Synthesis

- Higher abstraction level (behavior).
- Generate hardware from C or another high level language.
- Faster time to market.
  - Faster implementation
  - Faster verification
- Several hardware implementation alternatives can be generated from one HL implementation.
- A HL model can be used to generate hardware which meet different performance requirements and resource constraints.

High level Synthesis

- Open source tools and commercial tools are available: RoCCC, Catapult-C, MathWorks HDLCoder



HLS Process

- Data Flow Graph Analysis
- Resource Allocation
- Scheduling

Data Flow Graph Analysis

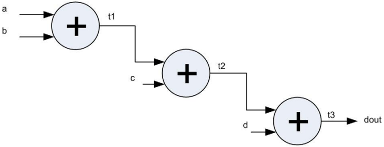
- High level synthesis starts by analyzing data dependencies in the code.
- This leads to a Data Flow Graph (DFG)
- Parts of code without dependencies can be executed in parallel

```
void accumulate(int a, int b, int c, int d, int &dout){
 int t1,t2;
 t1 = a + b;
 t2 = t1 + c;
 dout = t2 + d;
}
```

Data Flow Graph Analysis

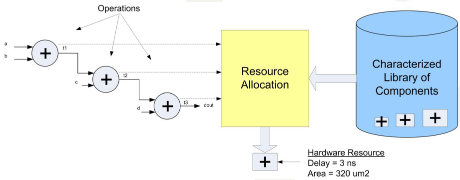
- High level synthesis starts by analyzing data dependencies in the code.
- This leads to a Data Flow Graph (DFG)
- Parts of code without dependencies can be executed in parallel

```
void accumulate(int a, int b, int c, int d, int &dout){
 int t1,t2;
 t1 = a + b;
 t2 = t1 + c;
 dout = t2 + d;
}
```



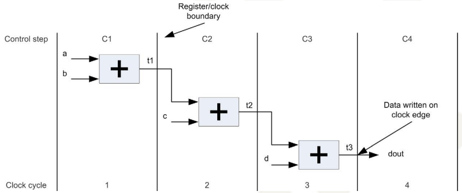
Resource Allocation

- Based on the assembled DFG, each operation is mapped onto a hardware resource.
- This process is called resource allocation.
- The implementation is annotated with both timing and area information. This is used during scheduling.



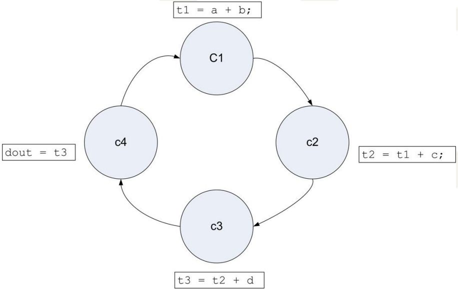
Scheduling

- HLS adds time to the design during the scheduling.
- Scheduling takes the DFG operations and decides when they are performed.
- Registers are added based on the target clock frequency.



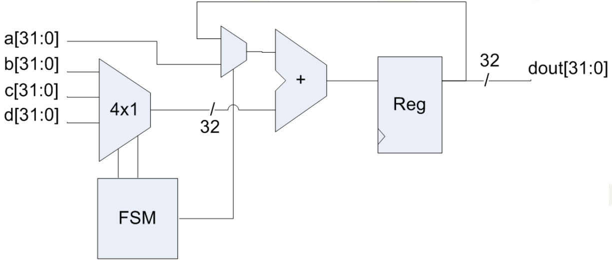
Scheduling

- A data path state machine (DPFSM) is created to control the scheduled design.
- In this design, four states are required to execute the schedule.
- These states are referred to as control steps or c-steps.



Scheduling

- The resulting hardware generated from the schedule will vary depending on the design constraints.
- Design constraints include resource allocation and performance.



Example

```c
unsigned int count_ones(unsigned int A)
{
  int i;
  unsigned count = 0;
  for(i=0; i<32; i++)
    if ((A & (0x00000001 << i)) > 0)
      count++;
  return count;
}
```

Not all code is suited for synthesis

- Need to keep in mind that the code ends up as hardware.
- Not all algorithms are suited for hardware implementations.
- Sequential code, control logic, large loops with function calls typically will not benefit much.

System Calls and Packages

- *stdio.h* functions such as *printf* or *cout*.
- *math.h* sin, cos and most math functions are not synthesisable.
- Assembly code is not synthesisable.
- System calls are generally not synthesisable.

Recursive functions

- Recursion are functions which calls itself.
- Used to write compact and efficient C code.
- Recursive functions executed on a CPU makes heavy use of the return stack, and are often unbounded.
- Typically a complete rewrite is required in order to get a iterative implementation which can be synthesised.

```
int *my_func (int *a) {
   *a+ = 20;
   if ( *a < 100 )
      return my_func (a);
   else
      return a;
}
```

Recursion is when a function calls itself.

Function pointers

- Function pointers are usually not supported.
- Needs to be changed to explicit function calls.

```
 int main() {
   void (*fp)(int);
   fp = func;
   fp(2);    // function pointer call
   func(2); // explicit function call
 }

void func(int arg) {
 printf("%d\n", arg);
}
```

Dynamic Memory Allocation

- Dynamic memory allocation is typically done with *malloc*
- Dynamic memory allocation is not synthesisable, and we need to use static memory allocations.

```
int static_mem[32];
int *dyn_mem = malloc(32, sizeof(int));
```

Unbounded Loops

- Loops without finite bounds.
- When the start value, stop value and the increment is constant and defined, the loop is bounded.
- A loop is not bounded when start, stop or increment is passed as a function parameter!

```
void unbounded_loop(int loop) {
  int i;
  for ( i = loop; i>=0; i--)
    statement;
}
```

Other restrictions

- Global variables for sharing data between functions are not supported.
- Problematic to pass pointers when using a CPU with MMU.

```
 int glob_var=5430;

 void func_0() {
   statement glob_var;
 }

 void func_1() {
   statement glob_var;
}
```

HLS Restrictions

- Restrictions in the HLS flow often requires rewriting C functions.
- C code targeting HLS is therefore less portable.
- Different HLS tools synthesis C code differently further decreasing portability.
- Programming code often require HLS tool specific adaptations.

Designing hardware with C

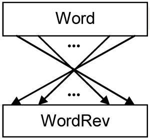
- We need to know how the tools transform C code into hardware in order to get efficient implementations.
- We need to know what type of code to avoid.
- HLS tools perform code transformations on the programming code when generating hardware implementations.
- Code transformations on different levels:
  - Bit-level
  - Instruction-level
  - Loop-level
  - Data-oriented

Bit-level transformations: bit reversing

- Software (a) implementation of bit reversing compared to a hardware implementation (b).

```
int Reverse(int Word) {
    int WordRev = 0;
        for(int i=0; i<32; i++) {
            WordRev |= (Word & 1);
            WordRev << 1;
            Word >> 1;
        }
    return WordRev;
}
```

(a)



(b)

Bit-level transformations: bit-width narrowing

- Variables are often defined with a greater dynamic range than needed.

Consider the example

```
for (int i=0;i<4;i++)
  statements
```

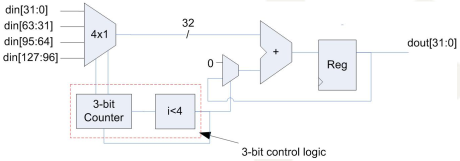
- On a CPU we use predetermined register widths.
- When implemented in hardware we allocate physical resources to the *i* variable.
- Bit-width narrowing can be determined by static analysis or profiling.

Bit-level transformations: bit-width narrowing

Consider the following example where bit-width narrowing is used to optimize the counter:

```
void accumulate4(int din[4], int &dout){
 int acc=0;
 for(int i=0;i<4;i++)
  acc += din[i];
 dout = acc;
}
```

Which results in the following hardware:

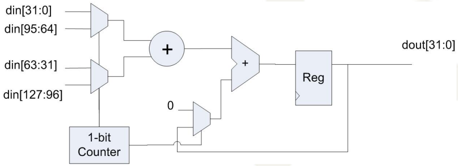


Loop-level trans.: Partial Loop Unrolling

In order to increase throughput consider the example:

```
void accumulate(int din[4], int &dout){
 int acc=0;
 for(int i=0;i<4;i+=2){
  acc += din[i];
  acc += din[i+1];
 }
 dout = acc;
}
```

Which results in the following hardware implementation:

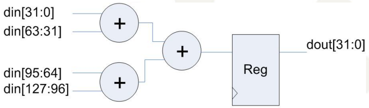


Loop-level trans.: Fully Unrolled Loop

Further increasing the throughput, consider the example:

```
void accumulate(int din[4], int &dout){
 int acc=0;
 acc += din[0];
 acc += din[1];
 acc += din[2];
 acc += din[3];
 dout = acc;
}
```

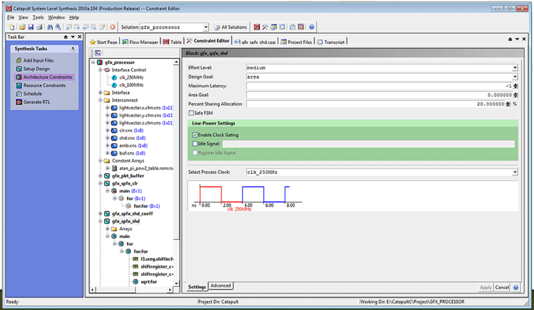
Results in a *balanced adder tree*:



Loop-level transformations

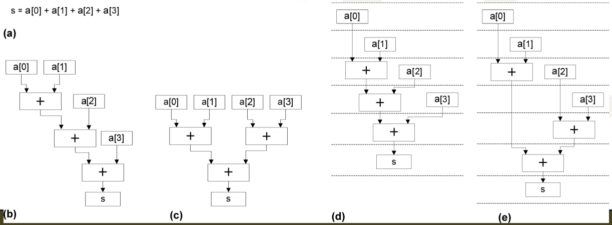
- HLS tools generate non-unrolled loops, partial unrolled loops, or fully unrolled loops out of a single implementation.

```
void accumulate4(int din[4], int &dout){
 int acc=0;
 for(int i=0;i<4;i++) acc += din[i];
 dout = acc;
}
```



Instruction-level trans: Tree Height Reduction

- Reduce the height of the a tree of operations by reordering them without changing the functionality.
- THR is applied to (b) which results in the tree shown in (c).
- (d) and (c) shows the scheduling when the data is stored in memory
- HLS tools will try to build a balanced tree structure out of related additions that can be scheduled in parallel.



Operation Strength Reduction (OSR)

- Replace an operation by a computationally less expensive one
- or a sequence of operations Example:

```
2<<1 == 2*2
```

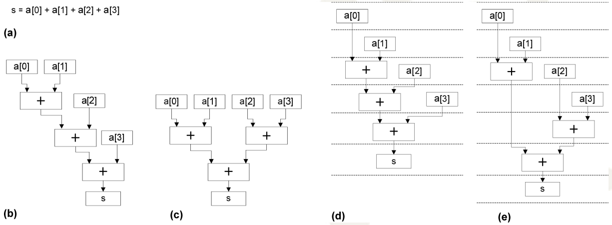
  - The multiplication is replaced with a simple shift. The shift only requires changes to the interconnections.

Data-Oriented Transformations

- Data-oriented transformations makes changes to the organization of data structures.
- Common transformations include:
  - Data distribution
  - Data replication

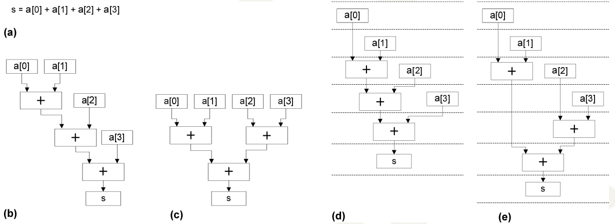
Data distribution transformation

- Partitions the data into many distinct internal memory units or modules.
- Increases throughput.
- Concurrent access.



Data duplication transformation

- Increases throughput.
- Concurrent access.
- Consistency issues when modifying data.



Other transformations

- Function inlining
  - Dedicated for each function invocation.

```
void accumulate() {
 accumulate4(din, dout);
 accumulate4(din2,dout2);
}

void accumulate4(int din[4], int &dout){
 int acc=0;
 for(int i=0;i<4;i++) acc += din[i];
 dout = acc;
}
```

Other transformations

- Function outlining
  - Increases resource sharing.
  - Reducing parallelism.

References

- Mentor Graphics The High Level Synthesis Blue Book
- Compiling for Reconfigurable Computing: A survey, Cardoso, Diniz, Weinhardt. DOI 10.1145/1749603.1749604