Software Programmable DSP Platform Analysis

Episode 7, Monday 19 March 2007, Ingredients

Software Pipelining

- Modern computers can execute parts of many different instructions at the same time (not only DSPs).
- C67x can execute up to 8 instructions at the same time.
- Software pipelining is a technique used to schedule instructions from loops (mostly) so that multiple iterations of the loop execute in parallel.

Data Dependence

If instruction *A* calculates a result that is used as an operand of instruction *B*, then *B* cannot be executed before *A* is finished.

Resource Constraints

- **Function unit**
  If there are k_{fu} multipliers (adders, etc) on the chip, then at most k_{fu} multiplication (additions, etc) instructions can execute at once.
- **Instruction Issue**
  The instruction issue unit can issue at most k_{ii} instructions at a time.
- **Register**
  At most k_r registers can be in use at a time (often split into files).

Resource Constraints in C67x (I)

An instruction scheduled on cycle i has the following constraints:

- **DP compare/ADDDP/SUBDP** (double precision float comparison)
  No other instruction can use the functional unit on cycles i and $i+1$
- **MPYI/MPYID/MPYDP**
  No other instruction can use the functional unit on cycles i,$i+1$,$i+2$, and $i+3$.
- ...

[SPRU189F, p.4-12]

Resource Constraints in C67x (II)

A *read/write hazard exists* when two instructions on the same functional unit attempt to read/write, to the refister file on the same cycle.

An instruction of the following types scheduled on cycle i has the following constraints:

- **2-cycle DP**
  - A single-cycle instruction cannot be scheduled on that functional unit on cycle $i+1$ due to write hazard on cycle $i+1$.
  - 2-cycle DP cannot be scheduled on that functional unit on cycle $i+1$ due to write hazard on cycle $i+1$.

[SPRU189F, p.4-12]

Resource Constraints in C67x (III)

- **4-cycle instruction**
  - A single-cycle instr. cannot be scheduled on that functional unit on cycle $i+3$ due to a write hazard on cycle $i+3$.
  - Multiplication cannot be scheduled on that functional unit on cycle $i+2$ due to write hazard on cycle $i+3$.
- **MPYI**
  - A 4-cycle instruction cannot be scheduled on that functional unit on cycles $i+4$,$i+5$,$i+6$.
  - MPYDP cannot be scheduled on that functional unit on cycles $i+4$,$i+5$,$i+6$.
  - Multiplication cannot be scheduled on that functional unit on cycle $i+6$ due to write hazard on cycle $i+7$.
- ...

[SPRU189F, p.4-13]

Resource Constraints in C67x (IV)

Double-Precision Floating-Point Addition

ADDDP (.unit) src1, src2, dst

stage	E1	E2	E3	E4	E5	E6	E7
read	*src1_l*	*src1_h*					
written						*dst_l*	*dst_h*
unit in use	.L	.L					

If *dst* is the source for **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP**, the number of delay slots can be reduced by 1. These instructions read the lower word of the source one cycle before the upper word.

[SPRU189F, pp.4-22/24]

We Do Need Automatic Scheduling!

With so many complex constraints, it quickly becomes a nightmare to find any nontrivial schedule for a given assembly program. Undoubtedly we need good automatic techniques for handling this.

`cl6x` incorporates an assembly optimizer containing a pipeline-aware automatic scheduler.

Manual techniques are given for highly fine-tuning your code with respect to pipeline in chapter 5 of Programmer's Guide. **[recommended reading]**

Loop Scheduling

Without Resource Bounds

For simplicity assume that

- Every instruction can be computed in 1 cycle.
- There are no resource constraints (only data constraints)
- The number of iterations in the loop is fixed (like with matrix multiplications)

Aiken-Nicolau Loop Pipelining

Loop Scheduling without Resource Bounds

1. Unroll the loop entirely.
2. Schedule each instruction from each iteration at the earliest possible time.
3. Find separated groups of instructions at given slope.
4. Coalesce the slopes.
5. Reroll the loop.

Example: A loop to be software-pipelined

$b \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a \leftarrow j \oplus b$
  $b \leftarrow a \oplus f$
  $c \leftarrow e \oplus j$
  $d \leftarrow f \oplus c$
  $e \leftarrow b \oplus d$
  $f \leftarrow U[i]$
  $g : V[i] \leftarrow b$
  $h : W[i] \leftarrow d$
  $j \leftarrow X[i]$

Trivial Linear Schedule

$b \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a \leftarrow j \oplus b$
  $b \leftarrow a \oplus f \| c \leftarrow e \oplus j$
  $d \leftarrow f \oplus c$
  $e \leftarrow b \oplus d \| f \leftarrow U[i] \| V[i] \leftarrow b \| W[i] \leftarrow d \| j \leftarrow X[i]$

- Assumption: no hardware constraints (just data dependencies)
- V, U, W and X are disjoint (commonly required)

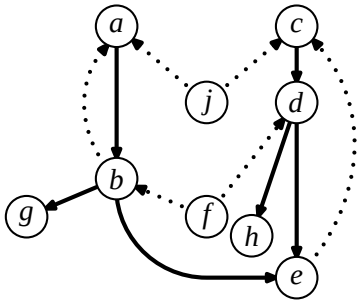
Aiken-Nicolau Loop Pipelining

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

After indexing accesses with iteration numbers.

Data-Dependence Graph

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

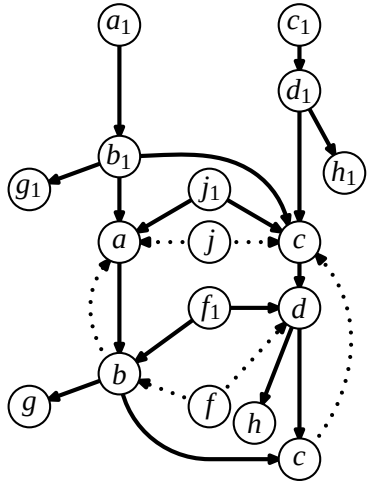


dotted: previous iteration deps

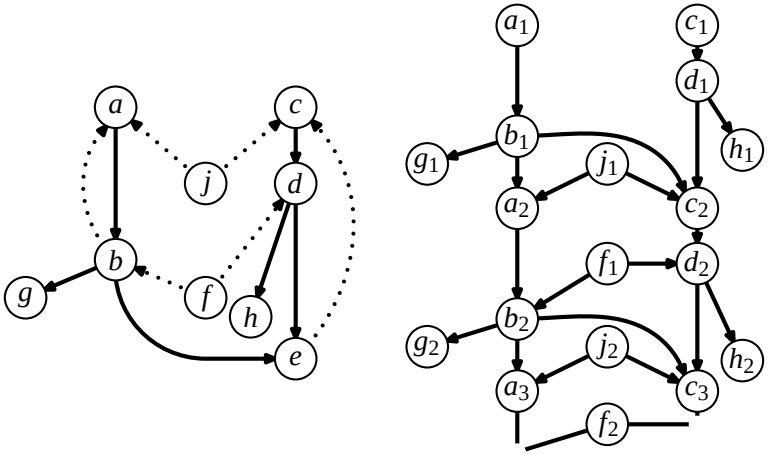
solid: current iteration deps

Data-Dependence Graph Unrolled

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$



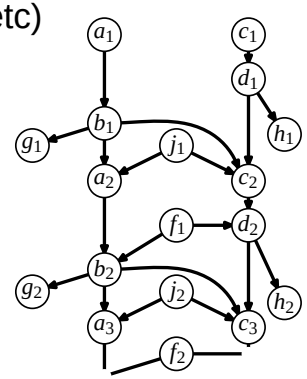
Data-Dependence Graph Unrolled



Schedule Completely Unrolled Loop

- Scheduling DAGs is easy: run each instruction as soon as all its predecessors have completed
- Do not take hardware limitations into account (unbounded concurrency, etc)

Cycle	Instructions
1	$a_1c_1f_1j_1f_2j_2f_3j_3$...
2	b_1d_1
3	$e_1g_1h_1a_2$
4	b_2c_2
5	$d_2g_2a_3$



Cycles/Iterations Tableau

Cycle	Instructions
1	$a_1c_1f_1j_1f_2j_2f_3j_3$...
2	b_1d_1
3	$e_1g_1h_1a_2$
4	b_2c_2
5	$d_2g_2a_3$

	1	2	3	4	5	6	iterations
1	*acfj*	*fj*	*fj*	*fj*	*fj*	*fj*	
2	*bd*						
3	*egh*	*a*					
4		*cb*					
5		*dg*	*a*				
6		*eh*	*b*				
7			*cg*	*a*			
8			*d*	*b*			
9			*eh*	*g*	*a*		
10				*c*	*b*		
11				*d*	*g*	*a*	
12				*eh*		*b*	
13					*c*	*g*	
14					*d*		
15					*eh*		

Cycles/Iterations Tableau: patterns

	1	2	3	4	5	6	iterations
1	*acfj*						
2	*bd*	*fj*					
3	*egh*	*a*					
4		*cb*	*fj*				
5		*dg*	*a*				
6		*eh*	*b*	*fj*			
7			*cg*	*a*			
8			*d*	*b*			
9			*eh*	*g*	*fj*		
10				*c*	*a*		
11				*d*	*b*		
12				*eh*	*g*	*fj*	
13					*c*	*a*	
14					*d*	*b*	
15					*eh*	*g*	

Cycles/Iterations Tableau: patterns

	1	2	3	4	5	6	iterations
1	*acfj*						
2	*bd*	*fj*					
3	*egh*	*a*					
4		*cb*	*fj*				prologue
5		*dg*	*a*				
6		*eh*	*b*	*fj*			
7			*cg*	*a*			
8			*d*	*b*			
9			*eh*	*g*	*fj*		new
10				*c*	*a*		body
11				*d*	*b*		
12				*eh*	*g*	*fj*	
13					*c*	*a*	
14					*d*	*b*	
15					*eh*	*g*	epilgoue

Optimal Schedule

- The optimal schedule can be found in Appel (Fig. 20.7, p. 483)
- It may require some more massaging if values of a given variable from several iterations are live at the same time (for example *f* in the example). This is shown on Fig. 20.8

- The schedule returned is guaranteed to be optimal for a machine that fullfills its hardware requirements (8 concurrent instructions in this case).
- $7 + (N - 4) + 5$ cycles, while the original loop scheduled totally sequentially required $9 * N$ cycles, while the trivial linear schedule (no instruction reordering) requires $4 * N$ cycles.
- For $N = 100$ the numbers are: 108, 900, and 400 respectively.
- On real machine it will surely take more cycles than 108.
- Software pipelining is difficult if there is control (branches) in the body of the loop.

Resource Bounded Loop Pipelining

Outline of the algorithm:

- Check if the body of the loop can be scheduled in Δ cycles.
- If this is possible finish.
- Otherwise increase Δ and retry.

Resource Bounded Loop Pipelining: Scheduling

- Take current instruction and schedule it at time $t = t_0$
- If this violates constraints than increase t
- Repeat this until t can be scheduled or $t = t_0 + \Delta$
- If no success then there is no schedule of lenght Δ (increase it).

Example

Same Program as Before, $\Delta = 3$

The machine can only perform one load instruction at a time.

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

An illegal schedule:

0		
1	$f_i \leftarrow U[i]$	$j_i \leftarrow X[i]$
2		

Example

Same Program as Before, $\Delta = 3$

We can try to move f_i one cycle back or forward:

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

Possibly extendable to a legal schedule:

0	$f_i \leftarrow U[i]$	
1		$j_i \leftarrow X[i]$
2		

0		
1		$j_i \leftarrow X[i]$
2	$f_i \leftarrow U[i]$	

Example

Same Program as Before, $\Delta = 3$

Actually we can go even one step further (backwards or forwards).

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

Possibly extendable to a legal schedule:

0		
1		$j_i \leftarrow X[i]$
2	$f_{i+1} \leftarrow U[i+1]$	

0	$f_{i-1} \leftarrow U[i-1]$	
1		$j_i \leftarrow X[i]$
2		

Example

Same Program as Before, $\Delta = 3$

Though not more. Two steps would clash with j again.

$b_0 \leftarrow V[0]$
for $i \leftarrow 1$ **to** N
  $a_i \leftarrow j_{i-1} \oplus b_{i-1}$
  $b_i \leftarrow a_i \oplus f_{i-1}$
  $c_i \leftarrow e_{i-1} \oplus j_{i-1}$
  $d_i \leftarrow f_{i-1} \oplus c_i$
  $e_i \leftarrow b_i \oplus d_i$
  $f_i \leftarrow U[i]$
  $g : V[i] \leftarrow b_i$
  $h : W[i] \leftarrow d_i$
  $j_i \leftarrow X[i]$

An illegal schedule:

0		
1	$f_{i+1} \leftarrow U[i+1]$	$j_i \leftarrow X[i]$
2		

0		
1	$f_{i-1} \leftarrow U[i-1]$	$j_i \leftarrow X[i]$
2		

Properties of the Algorithm

- The algorithm does such search in a greedy manner for consecutive instructions
- Typically it will start with instructions that are most dependent on other instructions, to increase a chance of earlier termination (relatively independent instructions are easy to schedule).

- The algorithm is not guaranteed to find an optimal solution
- It is not even guaranteed to find a solution of size Δ when one exists.
- It may also find a schedule for which a register allocation fails (and then it needs to be run again, as spills change scheduling)
- It may not terminate even! So often in practice it stops examining a given Δ after some time, and tries the next one.
- The consolation is that it is reported to work very well in practice.

Sometimes Cl6x Gives up on Pipelining

- If a value is live in a register for more than the number of cycles in the loop
- If the loop contains very complex conditions in the body.
- If the loop contains function calls (other than intrinsics or inlined functions)
- If the loop contains break statements (rewrite to use ifs).

: source Programmer's guide p. 2-54

“If” statements can only be used around the code that updates memory and around variables whose values are calculated inside the loop, but only used outside the loops (we know that this is keeping the induction variables linear...).

: source Programmer's guide p. 2-54

Thanks for Attention.