Memory Alias Disambiguation on the TMS320C6000™

George Mock — Software Development Systems

ABSTRACT

This application report is a tutorial and practical treatment on the problem of memory alias disambiguation on the TMS320C6000™ DSP. If you write C6000™ DSP linear assembly or hand-coded assembly, you will gain direct practical knowledge and advice on how to use the tools to handle this problem. If you write in C/C++, you will gain insight into how the compiler handles this problem, as well as some practical advice.

The keywords to keep in mind are: memory aliases, dependence graphs, instruction scheduling.

Contents

List of Tables

1 Overview

Memory alias disambiguation analyzes whether a dependence, through a memory location, exists between a given pair of instructions. Dependences between instructions are then used to determine the fastest legal schedule of those instructions.

This application report begins by covering the topic of dependence. Next is a description of how dependences are represented in dependence graphs. These concepts are then extended to cover loops. Then, it addresses how dependence affects instruction scheduling. Next, the term memory alias disambiguation is introduced.

The focus then shifts to how the tools, particularly the assembly optimizer, handle memory alias disambiguation. However, if you write hand-coded assembly, you will find some useful concepts in these sections. Several detailed examples are presented.

Two final sections discuss how the C/C++ compiler handles memory alias disambiguation, and the differences between memory alias disambiguation and memory bank conflict detection.

Note that this application report describes the C6000 code generation tools for release 2.10 or greater.

2 Background

2.1 Data Dependence Between Instructions

One dictionary definition of dependence is “the state of being determined, influenced, or controlled by something else”. In the world of software, the objects being influenced can be modules of code, specific functions, blocks within functions, individual statements, data structures, variables, etc. Further, the relationship can be interdependent, for example, two objects can depend on each other. This application report refers to only one kind of dependence relationship: the data dependence between individual assembly language instructions.

At this level, dependence is evaluated between pairs of instructions. Two instructions have a dependence when they reference (read or write) the same machine resource, for example, register, memory location, status bit, and so forth. A dependence is characterized by the following pieces of information:

- The first instruction
- The second instruction
- The resource both instructions reference
- The first instruction reference - read or write?
- The second instruction reference - read or write?

This information is summarized in the following table. The entries in the table are the formal name for that form of dependence.

Table 1. Dependence Table

		Instruction 2 Reference	
		Read	**Write**
Instruction 1 Reference	**Read**	Input	Anti-
	Write	Flow	Output

Flow dependence is the most common and intuitive form of dependence. In this relationship, one instruction writes an output which a following instruction reads as an input. For example:

```
I1:   ADDK  10,A2       ; writes output to A2
I2:   STW   A2,*A3      ; reads input from A2
```

Instruction `I1` writes an output in the register `A2`, and instruction `I2` reads `A2` as an input.

Anti-dependence is less common than flow dependence, but it is no less important. In this relationship, one instruction reads a resource as an input, and a following instruction writes a result to that same resource. For example:

```
I3:   STW   A3,*A4      ;reads A4
I4:   ZERO  A4          ;writes A4
```

Instruction `I3` reads `A4` for a data address, while instruction `I4` clears out `A4` for some later computation.

Note a key difference from flow dependence: the anti-dependence exists because of the reuse of the resource, and not because of a transfer of actual data. One easy way to remove an anti-dependence is to choose a different resource in the second instruction. In this example, instruction `I4` could use the register `A5` instead:

```
I3:   STW   A3,*A4      ;reads A4
I4:ZERO  A5             ;writes A5 ==> no anti-dependence
```

Since anti-dependence through a register is so easy to avoid, it is less common. However, anti-dependence through a memory location is usually not as easy to rewrite.

Output dependence is also not very common. One example is using a register to pass a value to a function. You will see a register load followed by a branch to a function which is known or presumed to overwrite that same register.

```
I5:   LDW   *A8,A4      ;load A4
I6:   B     func        ;branch to func ==> overwrites A4
```

The relationship between these two instructions is an output dependence.

Input dependence is common, but it is usually ignored. One exception is accessing memory mapped peripherals. In that case, reading a memory location can trigger a side effect such as incrementing register, or starting a memory block transfer. You generally want to recognize a dependence between any instruction which triggers such side effects and any other memory reference.

The term independent is used to describe two instructions which do not reference any of the same resources. Note the difference between the terms independent and anti-dependence.

2.2 Dependence Graphs

Dependence graphs are used to represent the dependences between a set of instructions.

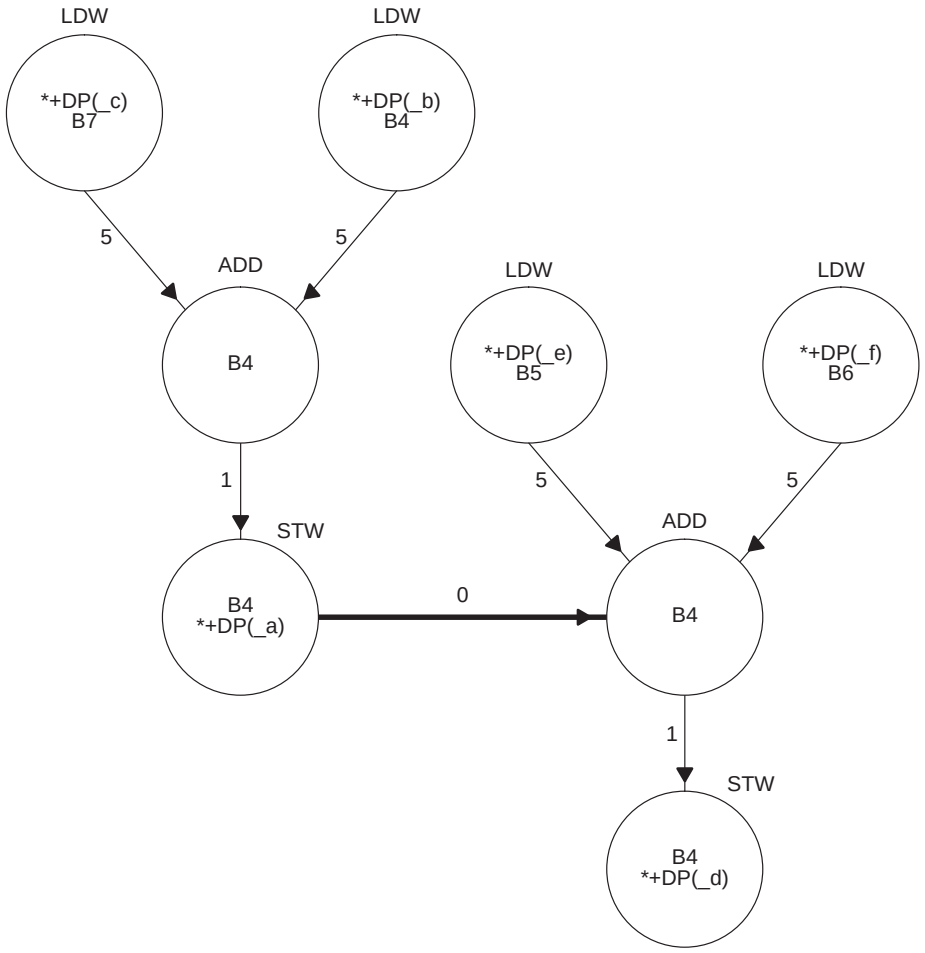
From the C fragment:

```
a = b + c;
d = e + f;
```

Here is hand modified compiler output which illustrates the serial instruction stream for those statements:

```
;-----------------------------------------------------------------------------
; 5 | a = b + c;
;-----------------------------------------------------------------------------
      LDW   *+DP(_b),B4 ; |5|
      LDW   *+DP(_c),B7 ; |5|
      ADD   B7,B4,B4    ; |5|
      STW   B4,*+DP(_a) ; |5|
;-----------------------------------------------------------------------------
; 6 | d = e + f;
;-----------------------------------------------------------------------------
      LDW   *+DP(_e),B5 ; |6|
      LDW   *+DP(_f),B6 ; |6|
      ADD   B6,B5,B4    ; |6|
      STW   B4,*+DP(_d) ; |6|
```

Here is the dependence graph:



The circles are called nodes, and the arrows connecting the circles are called edges. There is one node for each instruction, and one edge for each dependence. Since instructions can have multiple dependences as both input and output, nodes in the graph can have multiple edges leading in and out.

With regard to a single edge, the node at the head of the arrow is termed the “parent” of the “child” node at the tail of the arrow.

The instruction is written immediately over the corresponding node.

For loads and stores, both operands are written inside the node. For other instructions, only the result operand is written, because the input operands are the result operands from the parent nodes.

The numbers next to the edges indicate how many cycles of pipeline latency you must wait for that result to be available to the child node. A latency of 0 indicates those instructions can be scheduled in parallel.

A common misconception is to imagine data flowing along the edges. That is true for the common case of flow dependence (thus the name). But note the edge from the first STW to the second ADD. That is an anti-dependence on B4. No data is flowing in this dependence. The dependence is based on the reuse of register B4. Anti-dependence is shown in the graph with a boldface arrow. All of the other edges are flow dependences. In flow dependence, the associated operand is always the last (or only) operand shown in the parent node. For anti-dependence, the associated operand is the first (or only) operand shown in the child node.

In other literature, a node may be called a vertex (vertices for plural), and an edge may be called an arc.

If you are accustomed to the dependence graphs that appear in Chapter 7 of the *TMS320C6000 Programmer's Guide* (SPRU198), you will notice some differences. The graphs are called dependency graphs. An edge is called a path. Only one operand is shown in load/store instructions, and anti-dependence is not addressed.

2.3 Data Dependence in Loops

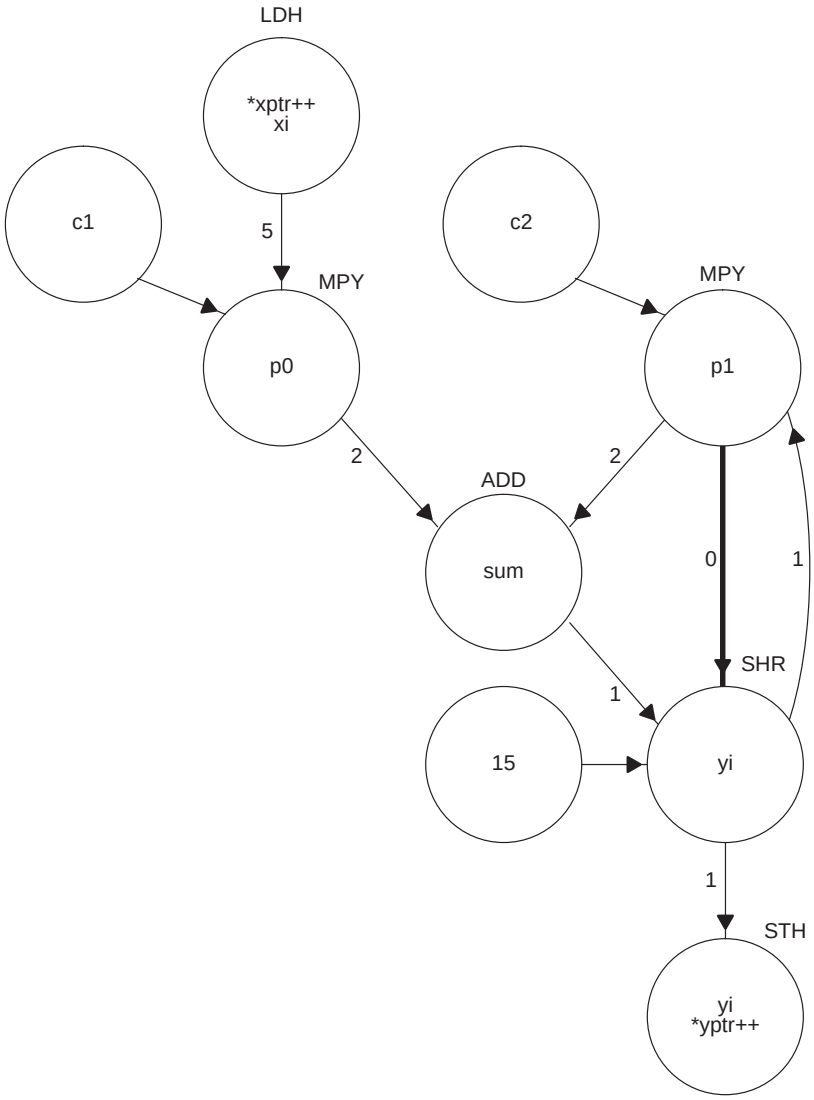
So far, we have only looked at relationships between instructions in a simple straight-line block of code. Considering dependence between instructions in a loop requires some extensions to those concepts.

A dependence graph for instructions in a loop looks the same, but there is a key difference. Each node, instead of representing one instruction, now represents every instance of that instruction in every loop iteration. The same is true of the edges.

When considering dependence graphs in straight-line code, you do not have to worry about the direction of the dependence, because it is always the same: from an earlier serial instruction to a later one. In loops, however, an instruction late in the loop can generate a result which is used, in the next loop iteration, by an instruction earlier in the loop. Such dependences are carried by the loop. In that case, the edge in the dependence graph goes the other direction. Here is a linear assembly code fragment:

```
loop:
LDH   *xptr++,xi
MPY   c1,xi,p0
MPY   c2,yi,p1  ; reads yi from prior iteration
ADD   p0,p1,sum
SHR   sum,15,yi ; writes yi for next iteration
STH   yi,*yptr++
; decrement and branch to loop
```

Here is the dependence graph:



Consider the instructions which reference `yi`. Note the flow dependence, carried to the next iteration by the loop, on yi from the SHR to MPY, because the SHR writes a value to `yi` which MPY reads. Also note the anti-dependence, not carried by the loop, on `yi` from the MPY to the SHR, because the MPY must read `yi` before SHR writes to it. Note how the two dependences are in opposite directions.

Nodes which are not loads and have no parents (c1, c2, 15) are either invariant in the loop or constants. No latency is shown by the edge since the operand is always available.

This application report only examines simple loops which contain no other loops. Data dependence in the presence of nested loops is beyond the scope of this report. With regard to the differences introduced by nested loops, the C6000 assembly optimizer capability and features, as well as this application report, work together to provide you with a conservatively safe solution. That is, the solutions we provide are generally optimal for simple loops, and safe, though sometimes less than optimal, for nested loops. The C/C++ compiler, on the other hand, performs very sophisticated dependence analysis on nested loops.

2.4 How Dependence Affects Instruction Scheduling

Instruction scheduling is solving the problem of choosing a schedule for a serial stream of instructions which satisfies two competing constraints: it preserves the computational effect of the serial instructions, for example, the code still works, and, it has the best performance.

Instruction scheduling algorithms are built around one central concept: while you do not have to honor the serial instruction order, you do have to honor the order imposed by the instruction dependences.

We can examine this concept at the C statement level. Take the C fragment presented earlier and simply swap the order of the statements:

```
d = e + f;
a = b + c;
```

It is obvious this will generate the same answer. The two statements are independent; they do not reference any of the same variables. Consider this fragment ...

```
x = y + z; /* #1 */
z = x + 1; /* #2 */
```

Obviously, if you reorder these statements, you will get a different answer. Consider the variable x. Statement 1 writes a value to x which statement 2 reads; a flow dependence on x. Consider the variable z. Statement 1 reads a value from z while statement 2 writes a value to z; an anti-dependence on z. Either dependence alone prevents reordering the statements.

It may be somewhat surprising that the forms of dependence, flow or anti- or output, all have the same effect on the statement order. In every case, the dependence constrains those statements to be in that order.

Input dependence is ignored with regard to scheduling; the order you read from memory usually does not matter. Instances where you may be concerned about input dependence include considering the effect on cache behavior, or accessing memory mapped peripherals.

These same ideas transfer directly to assembly language instructions. Instructions which are independent can be reordered, instructions which have one or more dependences cannot be reordered. Further, the latencies associated with the dependences must be honored.

Since dependences force instruction orderings, it follows that fewer dependences mean fewer constraints on instruction orderings. Put another way, fewer dependences mean more degrees of freedom in choosing an instruction schedule. On a chip architecture like the C6x, which has many opportunities for parallelism in combination with a deep pipeline, you can never have too much freedom in choosing an instruction schedule.

The details of how instruction scheduling algorithms really work is also beyond the scope of this application report. But here is the compiler generated schedule for the original C fragment presented earlier:

```
      LDW   .D2T2 *+DP(_b),B4 ; |5|
      LDW   .D2T2 *+DP(_c),B7 ; |5|
      LDW   .D2T2 *+DP(_f),B6 ; |6|
      LDW   .D2T2 *+DP(_e),B5 ; |6|
      NOP   3
      ADD   .L2   B7,B4,B4    ; |5|
      STW   .D2T2 B4,*+DP(_a) ; |5|
||    ADD   .L2   B6,B5,B4    ; |6|
      STW   .D2T2 B4,*+DP(_d) ; |6|
```

Note how the instructions from the two C statements are interspersed. The load statements are scheduled early, to better hide the latency of a load. The rest of the instructions are scheduled as soon as the latencies of the instructions they depend on are satisfied. Use the dependence graph from the earlier section as a guide.

2.5 Memory Alias Disambiguation Defined

Someone has an alias when he or she is known by two names. Nicknames are aliases. You might have had a friend in grade school who hated it when his mother called him “Thomas”, because you and all his friends called him by his preferred alias: “The Tomster”. Although “Thomas” and “The Tomster” are distinct names, they both refer to the same person.

This concept is analogous in computer programs. When there are two (or more) different ways to reference a memory location, we say there are aliases to that memory location.

Given this linear assembly fragment:

```
      I7:   LDW         *A4,A2
      ...                     ; other instructions
      I8:   STW         A3,*A6
```

do A4 and A6 reference the same memory location or not? If they do, they are memory aliases to that memory location. If they are memory aliases, then these two instructions have an anti-dependence between them; the read must occur before the write. In the instruction schedule, this dependence means those instructions must remain in that order.

On the other hand, if A4 and A6 do not reference the same memory location, they are not memory aliases. The instructions are independent. In the instruction schedule, these instructions can safely be placed in any order.

Note that unlike an anti-dependence on registers, there is no way to rewrite these instructions to remove the anti-dependence.

How can you determine whether *A4 is an alias for *A6 or not? Given the information we have here, you cannot. Thus, we call this an ambiguous alias. Maybe it is alias, maybe it is not.

Memory alias disambiguation, then, is the process of determining whether any given pair of memory references are aliases to one another. The outcome of that process is one of three states:

State	Means
Not aliases	Instructions are independent.
Are aliases	Instructions have a dependence between them.
Still ambiguous	Tool convention or user information is needed.

If a dependence is found, it imposes an ordering on the instruction schedule.

3 Tools Solution

3.1 Overview of the Assembly Optimizer Solution

In a few cases, the assembly optimizer attempts to automatically disambiguate as many memory references as possible. For all remaining memory references, the default is to presume they may access the same memory location, i.e. they are aliases. While that presumption is safe for all input, it is usually too conservative. So, a command line switch (`-mt`) and a function level directive (`.no_mdep`) can reverse that presumption, i.e. presume that any ambiguous aliases do not access the same memory location. If you have a small number of possible aliases in your code, you can use an additional directive (`.mdep`) to mark those instruction pairs. This is the recommended approach.

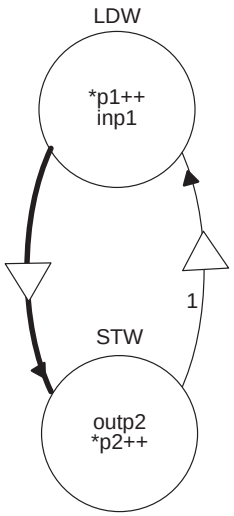
3.2 Default Presumption is Pessimistic

The default presumption, any ambiguous alias must be an alias, is a worst case, or pessimistic, assumption. While it is common to have instructions in your linear assembly which possibly access the same memory location, it relatively rare for that possibility to come true. Still, this pessimistic assumption is key to giving you the ability to balance correct handling of memory aliases with good performance.

The pessimistic assumption can have a drastic effect on software pipelining. Many linear assembly loops fit this general form:

```
loop:
I11:  LDW   *p1++,inp1
                          ; compute something into outp2
I12:  STW   outp2,*p2++
;decrement and branch to loop
```

Under the default assumption, `p1` and `p2` may reference the same memory location. This means two more dependence edges are added to the dependence graph: an anti-dependence edge between `I11` and `I12`, and a flow dependence edge between `I12` and `I11.`



When a dependence edge is associated with a memory reference, and not a register operand, you will see a triangle imposed over the edge.

These dependences mean those instructions must remain in that order for every loop iteration. Because these are the first and (nearly the) last instructions in the loop, and they cannot be moved past each other, software pipelining is all but completely disabled.

However, in many cases, `p1` and `p2` point to completely different arrays, and thus never reference the same memory location.

3.3 Change the Default Presumption to Optimistic

There are two methods for changing the presumption to the optimistic view that ambiguous memory aliases never access the same location. You can use a command line option:

```
cl6x –mt ...
```

Or, you can use a function level directive:

```
.no_mdep
```

The command line option affects every function in your linear assembly file. The `.no_mdep` directive can only appear within the `.(c)proc/.endproc` block of a linear assembly function, and affects only that function.

If you are certain you have no memory aliases in your code, then switching to the optimistic assumption is all you need to do. This will be a common case. If you ever do have a memory alias in your code, now you know how to handle it: get this application report out again.

Many users will want to switch to the optimistic assumption, except for a small number aliases they know about in their code. If that is you, the solution is to switch to the optimistic assumption, and then use the `.mdep` directive to mark those few aliases you have.

3.4 Using .mdep to Mark Aliases

Marking an instance of a memory alias is a two step process. First you attach symbolic names to your memory references in the linear assembly stream:

```
LDW   *p1++{ld1}, inp1  ; name memory reference ”ld1”
STW   outp2, *p2++{st1} ; name memory reference ”st1”
```

The names in the “{}” are assembly symbols like any other. You cannot use the same symbol as a memory reference name and a symbolic register. Then you note the specific memory dependence:

```
.mdep ld1,st1
```

This means whenever `ld1` references some memory location X, at some later time in code execution, `st1` may also reference location X. This is equivalent to adding an edge between these two instructions in the dependence graph. In terms of the instruction schedule, these instructions must remain in that order. The `ld1` reference must always occur before the `st1` reference.

Recall how the direction of a given dependence is important when considering loops. The direction implied by `.mdep` is from the first named memory reference to the second; in this case, from `ld1` to `st1`. The opposite direction, from `st1` to `ld1`, is not implied.

4 Examples of Memory Alias Disambiguation

4.1 How .mdep Affects Instruction Scheduling

The following are some complete examples. This example illustrates how an `.mdep` may, or may not, affect the instruction schedule. It also shows how the direction of a dependence, as indicated by the order of the operands to the `.mdep` directive, affects the instruction schedule.

Full understanding of all the examples presumes an understanding of the general concepts of software pipelining. For background information on software pipelining, see Chapter 7 of the *TMS320C6000 Programmer’s Guide* (SPRU198).

This linear assembly function is adapted from the weighted vector sum example. A typical call to this function could look like:

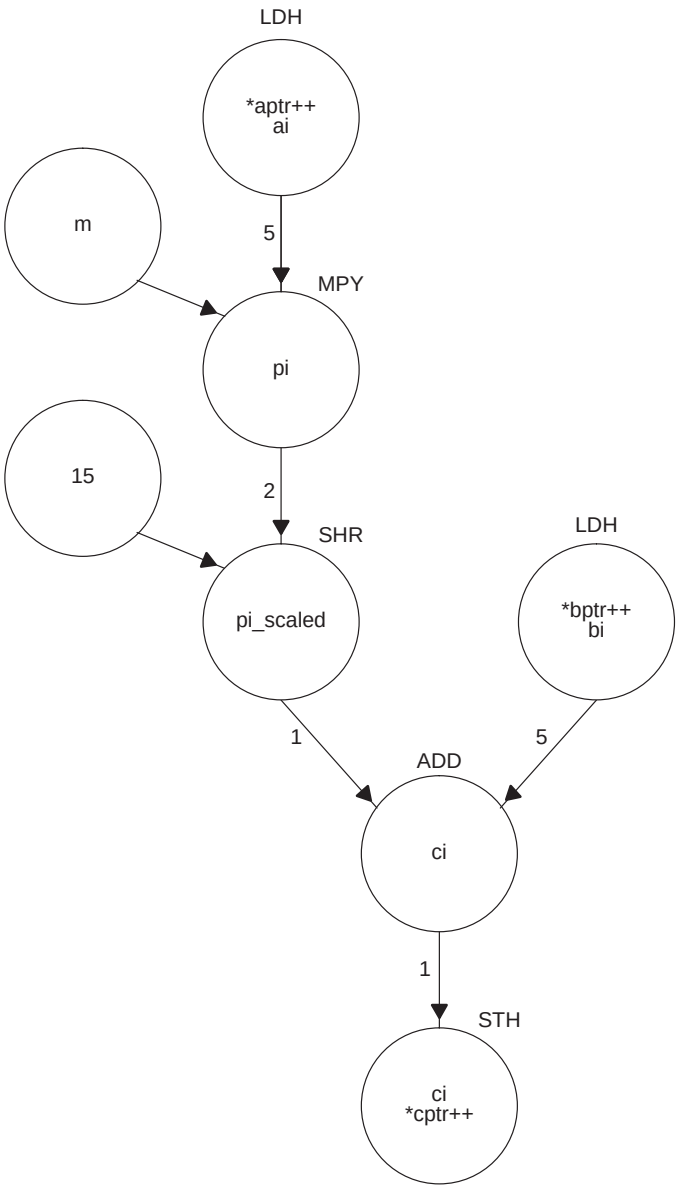
```
.call wvs(a, b, c, m)
```

Here is the source:

```
wvs:  .cproc aptr, bptr, cptr, m
      .reg  cntr, ai, bi, pi, pi_scaled, ci
      MVK   100,cntr
      .no_mdep       ; presume no memory aliases
loop: .trip 100
      LDH   *aptr++,ai
      LDH   *bptr++,bi
```

```
      MPY   m,ai,pi
      SHR   pi,15,pi_scaled
      ADD   pi_scaled,bi,ci
      STH   ci,*cptr++
[cntr]SUB   cntr,1,cntr
[cntr]B     loop
      .endproc
```

Here is the dependence graph (without the decrement and branch instructions):



The assembly optimizer generates a 2-cycle loop for this code, which is optimal for this input.

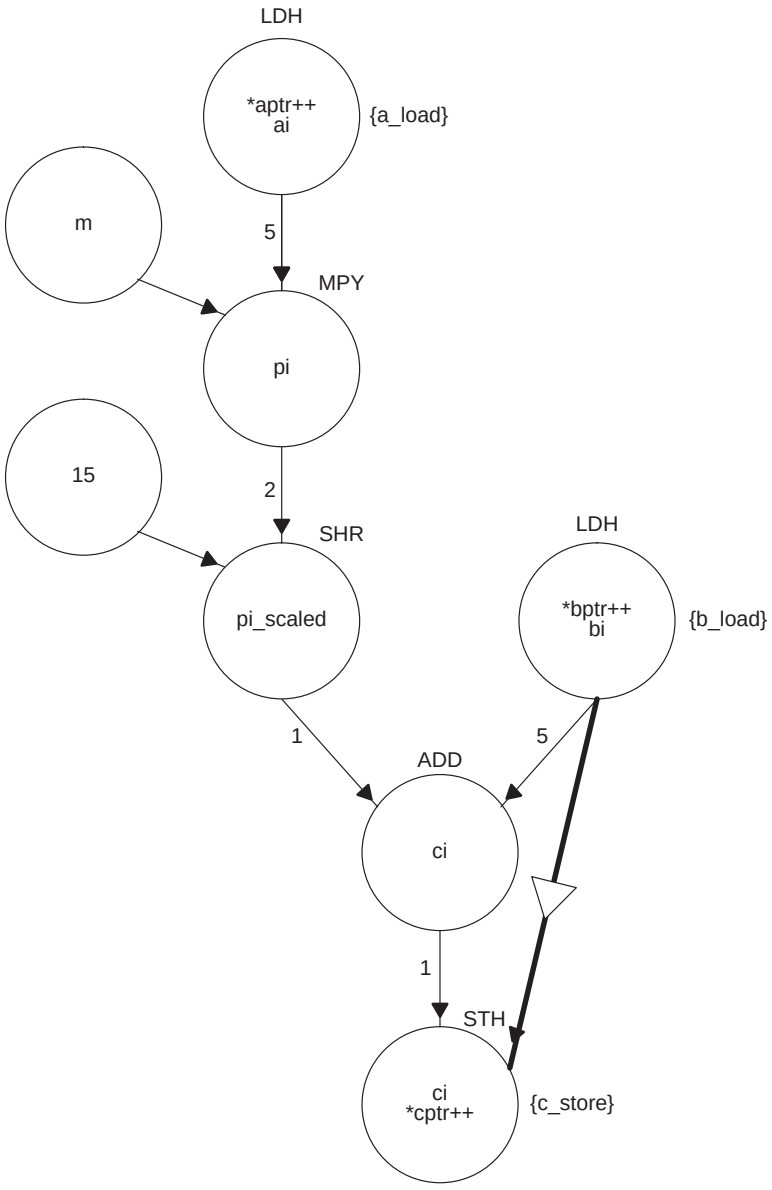
Suppose you know some calls to `wvs` pass the same array as the `b` input array and the `c` output array, just to save some space:

```
.call wvs(a, b, b, m)
```

Every loop iteration reads an element from the `b` array, and then immediately writes a result back to that same element. The correct way to model that is:

```
wvs:  .cproc aptr, bptr, cptr, m
      .reg  cntr, ai, bi, pi, pi_scaled, ci
      MVK   100,cntr
      .no_mdep             ; presume no memory aliases
      .mdep b_load,c_store ; except this one
loop: .trip 100
      LDH   *aptr++ {a_load},ai
      LDH   *bptr++ {b_load},bi
      MPY   m,ai,pi
      SHR   pi,15,pi_scaled
      ADD   pi_scaled,bi,ci
      STH   ci,*cptr++ {c_store}
[cntr]SUB   cntr,1,cntr
[cntr]B  loop
      .endproc
```

Here is the dependence graph:



Note the addition of the anti-dependence memory edge between the `b_load` and the `c_store`. The assembly optimizer still generates a 2-cycle loop for this code; the addition of the `.mdep` makes no difference because there is already a chain of flow dependences, through registers, from `b_load` to `c_store`. That chain of dependences imposes an ordering on the instructions in the chain. So, the instruction ordering constraint imposed by the anti-dependence edge is no different than the constraints already imposed by the flow dependence chain. Therefore, the instruction schedule doesn't change.

Or, you can think about it strictly in terms of the new anti-dependence memory edge. It means every time b_load references memory location X, at some *later* time in execution, c_store may also reference location X. This means that each b_load must occur before each c_store. More importantly, it also means each c_store does *not* have to occur before each b_load. So, the b_load for the next loop iteration can start before the c_store from the previous iteration finishes. Here is an illustration of the software pipeline where each iteration is in a separate column, and instructions which can run in parallel are on the same horizontal line:

Iteration n	Iteration n+1
...	
LDH b_load	...
...	LDH b_load
STH c_store	...
	STH c_store

The software pipeline was structured like that before the .mdep. So, no change.

While it makes for a contrived example, consider what happens if you call wvs like this:

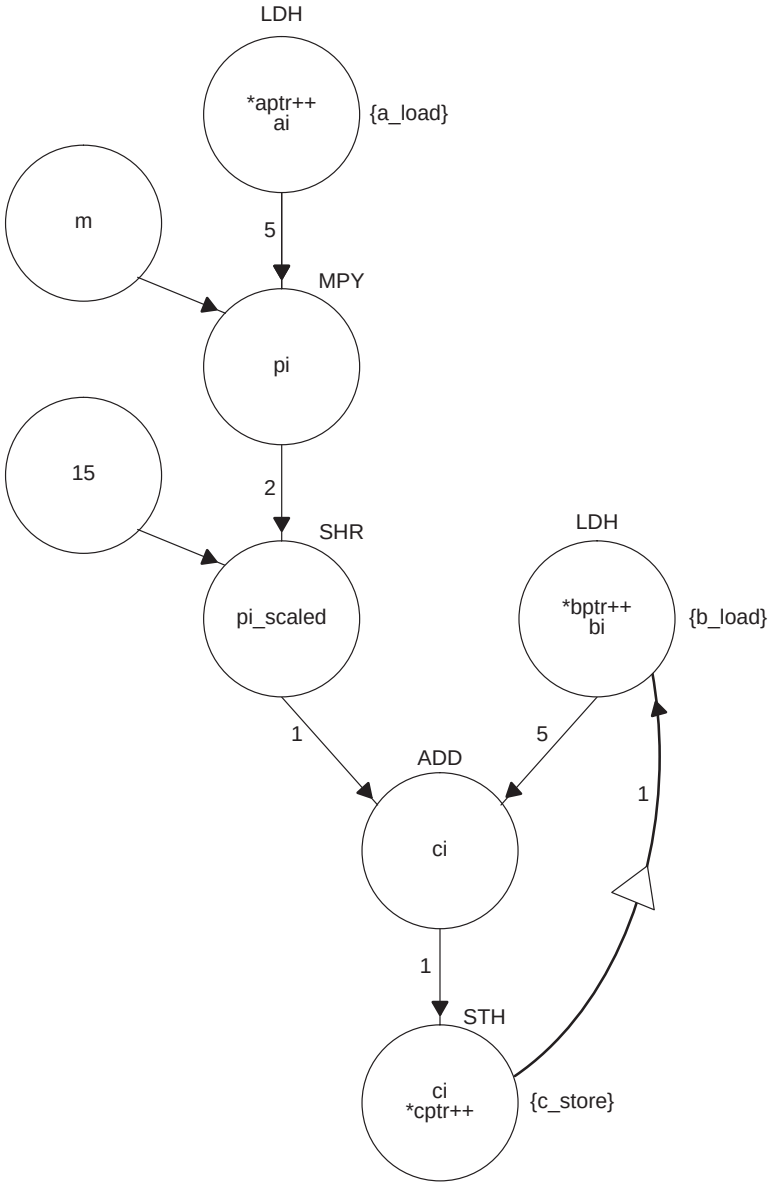
```
ADD   b,2,c        ; c points to b[1]
.call wvs(a, b, c, m)
```

c_store writes its result to an array element which b_load reads on the next loop iteration. Here is the correct way to model that:

```
.no_mdep                ;presume no memory aliases
.mdep c_store,b_load    ;except this one
<exactly as before>
```

Note the .mdep is the same as the previous example, except the operands are reversed.

Here is the dependence graph:



Now, instead of the anti-dependence memory edge, there is a flow dependence memory edge from `c_store` to `b_load`. Note this dependence is carried by the loop. Now the assembly optimizer generates a 7 cycle loop.

Recall the chain of flow dependences, on registers, from the `b_load` to the `c_store`. Now that chain is extended, and carried by the loop, to the `b_load` for the next iteration. Before you can start that `b_load` for the next loop iteration, you have to wait for the `c_store` from the present iteration to finish. Here is how the software pipeline looks:

Iteration n	Iteration n+1
...	
LDH b_load	
...	
STH c_store	...
	LDH b_load
	...
	STH c_store

As you can see, the direction, as implied by the operand order, is a very important characteristic of an .mdep.

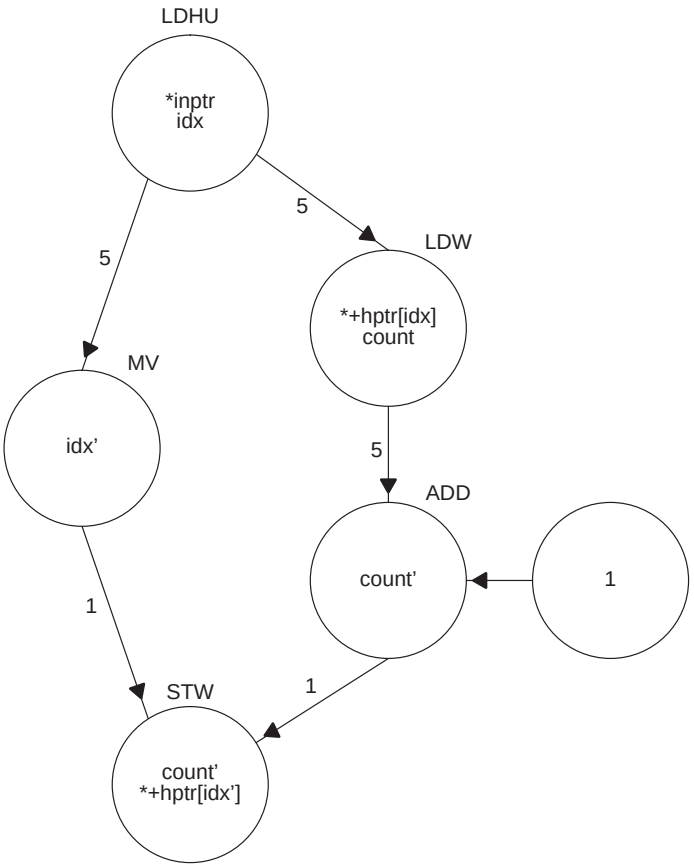
4.2 Handling Indexed Addressing Mode

Indexed addressing, e.g. *+A4[A5], typically means you are accessing memory without any clear pattern.

Here is an example of how to handle this case.

```
histogram: .cproc inptr, hptr, len
      .reg  idx, count
      .no_mdep       ; no memory aliases
loop: .trip 8
      LDHU  *inptr++,idx
      LDW   *+hptr[idx],count
      ADD   count,1,count
      STW   count,*+hptr[idx]
[len] SUB   len,1,len
[len] B  loop
      .endproc
```

Here is the dependence graph:



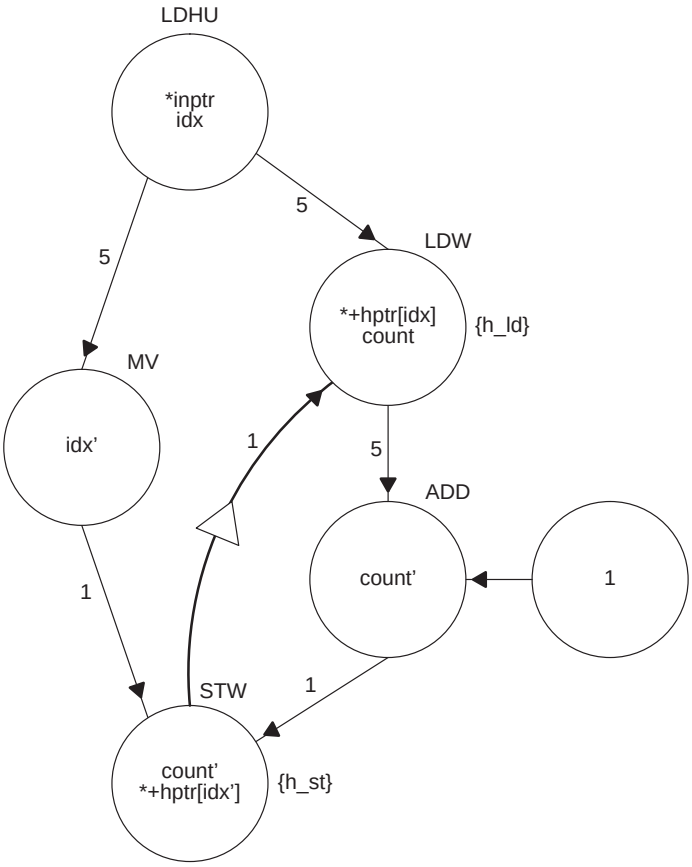
Note the assembly optimizer splits the lifetime of the `idx` register by adding the `MV` instruction; the new variable is shown as `idx'`. The lifetime of `count` is similarly split at the `ADD` instruction.

The assembly optimizer generates a 2 cycle loop, but it will not work. This loop is computing, into the array `hptr`, a histogram of all the values in the array `inptr`. What if the value 10 occurs in the `inptr` array two times in a row? In that case, the location `*hptr[10]` is incremented on successive loop iterations. Look at the dependence graph. Do you see a dependence edge from the `STW` to the `LDW`? That is the problem. The `LDW` for the next loop iteration has permission to get started without waiting for the `STW` from the previous loop iteration, which it does. To fix this situation we add the `.mdep:`

```
histogram:  cproc inptr, hptr, len
      reg   idx, count
      no_mdep                 ;no memory aliases
      mdep h_st, h_ld         ;except this one
loop  trip  8
      LDHU  *inptr++,idx
      LDW   *+hptr[idx] {h_ld},count
      ADD   count,1,count
      STW   count,*+hptr[idx] {h_st}
```

```
[len]    SUB    len,1,len
[len]    B  loop
         .endproc
```

Here is the dependence graph:



Note the flow dependence memory edge, carried by the loop, from the STW to the LDW. Now the assembly optimizer generates a 7 cycle loop. Much slower, but it works.

Do you need the .mdep in the other direction, from the h_ld to the h_st? If you simply want the code to run, and you do not care why, then the answer is no. Because of the chain of flow dependences on registers from h_ld to h_st, this ...

```
no_mdep              ; no memory aliases
.mdep h_st, h_ld     ; except this one
mdep  h_ld, h_st     ; and this one
```

does not change the instruction schedule. But the dependence actually does exist, so it is advisable to add this .mdep because it makes the code self-documenting.

In the face of indexed addressing, you may be tempted to just rely on the default pessimistic assumption. Be careful. In this case, that will hand you a 13-cycle loop. Under the pessimistic assumption a dependence is recognized from the STW (at the bottom of the loop) to the LDHU (at the top of the loop). That means the load at the top of iteration n+1 cannot start until the store at the end of iteration n is finished.

4.3 Peripherals Access Example

Recall that you cannot override any automatic disambiguation performed by the assembly optimizer. If it can determine that two memory references (must/must not) access the same memory location it (will/will not) recognize a dependence between the associated instructions. This is true despite any command line options or `.mdep` directives which may be in effect. This means there is no way to guarantee the assembly optimizer will use a particular pattern of access to memory. In general code, this is preferable behavior. But it can be an issue when you consider code which accesses memory mapped peripherals. Here is an example:

```
;base of multi-channel buffered serial port 0
MCBSP0_BASE .set 0x018C0000
mcbsp0_dxr: .cproc
      MVKL  MCBSP0_BASE,B4    ;load base of McBSP0 regs
      MVKH  MCBSP0_BASE,B4
      ...
      STW   B5,*+B4(0x10)     ;init XCR for transfer
      STW   B6,*+B4(0x4)      ;transfer word through DXR
      ...
      .endproc
```

It is clear that the two STW memory references are not accessing the same memory location; they are using the same base register with different offsets. So, even if you use:

```
.mdep wrt_xcr,wrt_dxr
STW         B5,*+B4(0x10) {wrt_xcr}
STW         B6,*+B4(0x4) {wrt_dxr}
```

the assembly optimizer may still reorder the writes. In general code, this is fine, and often an improvement. But when accessing peripherals like a serial port, or whenever writing to a memory location can trigger a side effect, reordering the memory references is wrong.

Presently, there are two ways to solve this problem. You can write the code in C, being careful to use the keyword “volatile” for any memory reference which has a side effect. Or, you can bypass the assembly optimizer by writing these routines in hand-coded assembly.

5 C/C++ Compiler and Alias Disambiguation

The C/C++ compiler provides several methods, both command line options and in the source, for addressing the problem of memory alias disambiguation. Having read this application report, you should now have a much better understanding of the issue. This section will briefly review each of these methods. For all the details, consult either the *TMS320C6000 Programmer’s Guide* (SPRU198) or the *TMS320C6000 Opimizing C Compiler User’s Guide* (SPRU187).

The compiler does a much better job of alias disambiguation than the assembly optimizer. C source provides much more information to work with. The default presumption on aliases which cannot be disambiguated is the pessimistic one: they are aliases.

Still, there are a few very esoteric cases of memory aliases which the compiler presumes do not occur. If your code violates those presumptions use:

```
cl6x -ma
```

On rare occasions, you may need it.

The command line option:

```
cl6x -mt
```

means something different to the compiler than it does to the assembly optimizer. As presented already, this option reverses the assembly optimizer's pessimistic assumption that memory references it cannot disambiguate must be aliases. To the compiler, this same option means several specific instances of memory aliases do not occur in your C/C++ code.

The command line options:

```
cl6x –pm –o3
```

have several effects, of which improved alias disambiguation is only one. These options work together to provide program level optimization. The `–pm` option combines all of your source files into one intermediate file, and `–o3` carries out the program level optimization on that intermediate file. Seeing all the functions at once yields optimization opportunities which generally do not occur otherwise. If the compiler can see all the calls to this function:

```
void foo(int *p1, int *p2)
```

it can easily determine that the same array is never passed in for `p1` and `p2`, and therefore `p1` and `p2` references are not aliases.

Correct use of the `const` and `restrict` keywords helps the alias disambiguation problem. The `const` keyword tells the compiler that the data object will not be modified, even by an alias. So, any `const` qualified memory read cannot possibly be an alias to a memory write. If an alias does modify a `const` object, that is a user bug. The `restrict` keyword tells the compiler that within the scope of a particular pointer, only the object pointed to can be accessed by that pointer.

6 Memory Alias Disambiguation versus Memory Bank Conflict Detection

If a C6000 execute packet (a set of instructions which execute in parallel) includes two references to memory, and both of those references are to the same memory bank, because each bank is single-ported memory, the pipeline stalls for one cycle while the second memory word is accessed. This is called a bank conflict, and it is obviously worth avoiding. The assembly optimizer provides a directive called `.mptr` for the purpose of solving this problem. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for all the details.

It is easy to confuse the topic of memory alias disambiguation with memory bank conflict detection. The terms sound similar. And they are both concerned with how memory references affect the instruction schedule. But there are some striking differences ...

	Alias Disambiguation	Bank Conflict Detection
The problem is ...	Whether two memory references access exactly the same location	Whether two memory references access the same memory bank
The answer affects ...	The ordering constraints imposed on the instruction schedule	Whether to schedule a pair of memory references in parallel
Get it wrong and ...	Your code does not work	The execute packet takes 1 cycle longer
Occurrences of ...	Memory aliases are relatively rare	Potential memory bank conflicts are common

You have to solve the problem of memory alias disambiguation before you can consider the problem of memory bank conflict detection. One of the presumptions of memory bank conflict detection is that the two memory references can be scheduled in parallel. That is true only if you have already determined the instructions are independent; they are not aliases to one another.

In your linear assembly, it is best to simply keep these problems, and their solutions, entirely separate. Enter your `.mdep` directives without any regard to your `.mptr` directives, and vice versa.

7 Summary

- Dependence is a relationship between two instructions which read or write the same machine resource.
- Dependence graphs represent the dependence between instructions. Nodes (circles) are instructions. Edges (arrows) are dependences. Data often flows along edges, but not always.
- In loops, nodes represent every instance of that instruction in every loop iteration, and dependence direction is important.
- Dependences force an ordering on the instruction schedule.
- Generally, the fewer the dependences, the better the schedule.
- Multiple references to the same memory location are called aliases.
- Aliases imply a dependence between the associated instructions.
- Memory alias disambiguation is the process of determining whether a pair of references are aliases, for example, whether a dependence is recognized between the instructions.
- The assembly optimizer automatically disambiguates a few references, then uses command line options and directives to disambiguate the remaining references.
- The default presumption for remaining aliases is pessimistic; they are aliases.
- The assembly optimizer command line option:

```
cl6x –mt ...
```

reverses the default presumption to optimistic; they are not aliases. It applies to all functions in the file.

- The function level directive ...

```
.no_mdep
```

also changes the presumption to optimistic, but applies only to the function which contains it.

- To mark a specific memory dependence, first annotate memory references ...

```
LDW   *p1++{ld1}, inp1  ; name memory reference ”ld1”
...
STW   outp2, *p2++{st1} ; name memory reference ”st1”
```

Then note the specific dependence:

```
.mdep ld1,st1
```

- You cannot force the assembly optimizer to recognize a dependence between instructions, which can be an issue when accessing memory mapped peripherals.
- The C/C++ compiler offers the user several methods for influencing the handling of memory aliases
- Do not confuse memory alias disambiguation with memory bank conflict detection. Solve the problems separately.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.