Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Explaining the Branch Target Buffer on the ADSP-TS101

Last modified: 2/02

Contributed By: C.L.

The following Engineer-to-Engineer note will discuss the branch target buffer (BTB) on the ADSP-TS101 TigerSHARC DSP. It will explain how the branch target buffer works, when it is advantageous to use it, and some of the errors that can occur because of it.

Introduction

The TigerSHARC can achieve its fast execution rate in part because of its eight-cycle deep pipeline. However, the drawback to this deep instruction pipeline is that when a branch instruction is executed, the pipeline must be flushed. The new instruction that is being branched to must then traverse the entire pipeline before it can be executed. This can cause a latency of three to six cycles. Using the BTB can reduce this latency to zero cycles.

The BTB is a 4-way set associative cache for branch instructions. This includes: interrupt returns, call returns, and computed jump instructions. It is 128 entries deep and uses a Least Recently Used replacement policy.

Each entry in the BTB has a TAG and a TARGET field. The TAG field stores the quad address of the instruction line that contained the branch instruction. The TARGET field stores the address that the program sequencer will jump to if the branch is taken.

How the BTB saves cycles

The first time a branch instruction with a true condition occurs in code, the BTB will not have an entry for it. Therefore, a new entry is placed, with the TAG being the value in the PC. There will be a penalty of two cycles taken at this point.

However, every time the same branch instruction is run into again, the value of the PC will match the value stored in the TAG field that was stored the last time. The program sequencer will see that there is a BTB hit, and instead of grabbing the next sequential instruction, it will grab the instruction at the address stored in the TARGET field. Therefore, if the branch condition ends up being true and the branch is taken, the correct instruction is already being sent through the pipeline. The cycle penalty for taking this branch is now reduced zero cycles.

When Is It Advantageous To Use the BTB?

Using the BTB will save the most cycles when the same branch instruction is executed over and over again, and each time the condition that controls the branch is true. For example, if you are in a loop that contains a branch instruction that is always taken, then the first time through the loop there will be a two-cycle penalty. Each subsequent time through the loop there will be zero cycle penalties. Compare this to the same code that does not use the BTB. Depending on whether the condition depends on the IALU or the compute blocks, there will be a 3 or 6 cycle penalty every time through the loop.

The worst situation in which to use the BTB is where the branch instruction is executed over and over again as before, but the condition is only true the first time. If the branch is taken the first time, then each subsequent time the branch will be predicted. A predicted branch that is not taken delivers the same 3 or 6 cycle penalty. Compare this to where there is no branch prediction. The first time through the loop, you will suffer from the 3 or 6 cycle penalty. But each additional time through the loop you will suffer zero penalties

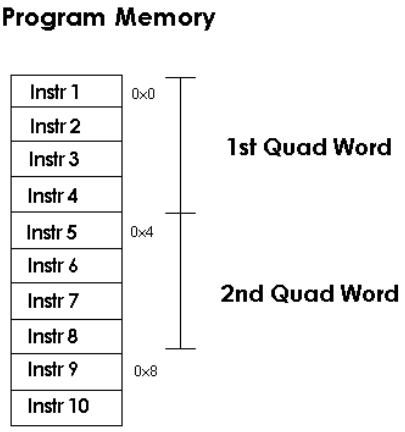
In general, if the branch is going to be taken more often than not, then it should be predicted. If the branch is not taken most of the time, then it should not be predicted. You can force a branch instruction to not be predicted by adding the (NP) suffix as shown below.

```
If jeq, jump 0x0000 (NP);;
```

Putting Two Branches in One Quad Word

One of the caveats of using the Branch Target Buffer is being aware of not putting two branch instructions in the same quad word in memory. The compiler will issue a warning saying "Detected two instruction lines with predicted jumps ending within 4 words" if it thinks this might be happening.

Understanding how to avoid this situation begins with understanding how instructions move from internal memory to the core. Every cycle, the DSP takes advantage of its 128-bit bus to grab four instructions from internal memory. The first of these four instructions always has an address that is quad aligned (it is a multiple of four).



Remember that the TigerSHARC can execute anywhere from one to four instructions per one instruction line. So while four instructions are being loaded into the core every cycle, those four instructions are not necessarily being executed in the same instruction line. To handle this situation, the instructions are stored in a five word FIFO called the Instruction Alignment Buffer (IAB). Here, they are aligned into the proper instruction lines as defined in the source code. For example, consider code structured as shown below.

```
Instr1;;
Instr2; Instr3; Instr4; Instr5;;
Instr6; Instr7;;
Instr8;;
```

In the first cycle, the first quad word of instructions is fetched from memory. This includes instructions 1, 2, 3, and 4. The first instruction line can be executed because it only contains Instr1. The second instruction line must wait for the next quad word of instructions to be loaded from memory since it needs Instr5. Instructions 2, 3, 4, and 5 are aligned into one instruction line in the IAB. The instructions for

the next two instruction lines are already loaded from internal memory and are ready for execution in the following cycles.

Understanding this, we are now ready to look into how it affects the operation of the BTB. The address that is stored in the TAG field of the BTB can only be a quad word address (0x0, 0x4, 0x8…). So if a branch instruction occurs in instruction 1, 2, 3, or 4, then the TAG that will be stored is for the quad word 0x0. If the branch instruction occurs in instruction 5, 6, 7, or 8, then the TAG that will be stored is for the quad word 0x4. It can easily be seen that there cannot be two branch instructions located within the same quad word location in memory. If Instr3 and Instr4 both were branch instructions, they would have the same TAG value of 0x0, and the BTB would not be able to discern between the two.

Also important to understand is that the BTB TAG field stores the quad word address of the last instruction of the instruction line that contains branch. For example, consider the code structure below.

```
Instr1;;
Instr2; Instr3; Branch(Instr4); Instr5;;
Instr6;;
```

Here the branch instruction (Instr4 in our memory diagram) is in the second instruction line of code. But even though Instr4 is in the quad word with address 0x0, this is not the address that is stored in the TAG field of the BTB. The program sequencer must load all the instructions that are executed in the same instruction line, so Instr5 must also be loaded. Since its quad word address is 0x4, this is the address that is loaded into the TAG field of the BTB.

Now, when the compiler gives the warning "detected two instruction lines with predicted jumps within 4 words", we can adequately understand how to solve the problem. The last instruction in an instruction line that contains a branch must be more than four memory locations from the last instruction of another line with a branch in it. For example, consider the code structure shown below.

```
Branch(Instr2); Instr3; Instr4; Instr5;;
Instr6; Branch(Instr7);
```

This will cause a compiler warning and an error in the BTB, since both instruction lines will try to load the quad address of 0x4 as their TAG. Changing to the code structure show below will solve the problem.

```
Branch(Instr2); Instr3; Instr4; Instr5;;
Instr6; Branch(Instr7); Instr8; Instr9;;
```

Now, the first instruction line's TAG will be quad address 0x4, and the second instructions line's TAG will be quad address 0x8. Not only has the problem been solved, but it also has been done without adding any extra instruction lines, which would lead to extra cycle penalties.