3. (20%) Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always 4 cycles and the buffer miss penalty is always 3 cycles. Assume 90% hit rate and 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed 2-cycle branch penalty? Assume a base CPI without branch stalls of 1.

---

4. (20%) List all the dependences (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop carried or not. Show why the loop is not parallel.

```
for (i=2;i<100;i=i+1) {
        a[i] = b[i] + a[i];     /* S1 */
        c[i-1] = a[i] + d[i]; /* S2 */
        a[i-1] = 2 * b[i];     /* S3 */
        b[i+1] = 2 * b[i];     /* S4 */
}
```

---

5. (20%) Let's consider the effects of interleaving memory and wide memory. Consider the following description of a computer and its cache performance:

Block size = 1 word
Memory bus width = 1word
Miss rate = 3%
Memory accesses per instruction = 1.2
Cache miss penalty = 64 cycles
Average cycles per instruction (ignoring cache misses) = 2

The performance of the memory is:

4 clock cycles to send the address
56 clock cycles for the access time per word
4 clock cycles to send a word of data

If we change the block size to 2 words, the miss rate falls to 2% and a 4-word block has a miss rate of 1.2%. What is the improvement in performance of interleaving two ways and four ways versus doubling the width of memory and the bus?

國立中山大學資訊工程學系博士班資格考
Computer Architecture Spring, 2011

1. Amdahl's Law implies that the ultimate goal of high-performance computer system design should be an enhancement that offers arbitrarily large speedup for all of the task time. Perhaps surprisingly, this goal can be approached quite closely with real computers and tasks. Section 3.5 describes how some branch instructions can, with high likelihood, be executed in zero time with a hardware enhancement called a branch-target buffer. Arbitrarily large speedup can be achieved for complex computational tasks when more efficient algorithms are developed. A classic example from the field of digital signal processing is the discrete Fourier transform (DFT) and the more efficient fast Fourier transform (FFT). How these two transforms work is not important here. All we need to know is that they compute the same result, and with an input of n floating-point data values, a DFT algorithm will execute approximately n^2 floating-point instructions, while the FFT algorithm will execute approximately $n \log_2 n$ floating-point instructions.

(10%) a. [12] <1.6> Ignore instructions other than floating point. What is the speedup gained by using the FFT instead of the DFT for an input of $n = 2^k$ floating-point values in the range $8 \le n \le 1024$ and also in the limit as $n \to \infty$?

(5%) b. [10] <1.6> When $n = 1024$, what is the percentage reduction in the number of executed floating-point instructions when using the FFT rather than the DFT?

(5%) c. [Discussion] <1.6> Despite the speedup achieved by processors with a branch-target buffer, not only do processors without such a buffer remain in production, new processor designs without this enhancement are still developed. Yet, once the FFT became known, the DFT was abandoned. Certainly speedup is desirable. What reasons can you think of to explain this asymmetry in use of a hardware and a software enhancement, and what does your answer say about the economics of hardware and algorithm technologies?

---

2. One use of saturating arithmetic is for real-time applications that may fail their response time constraints if processor effort is diverted to handling arithmetic exceptions. Another benefit is that the result may be more desirable. Take, for example, an image array of 24-bit picture elements (pixels), each comprised of three 8-bit unsigned integers, representing red, green, and blue color brightness, that represent an image. Larger values are brighter.

(10%) a. [10] <2.8> Brighten the two pixels E5F1D7 and AAC4DE by adding 20 to each color component using unsigned arithmetic and ignoring overflow to maintain a fixed total instruction-processing time. The values are given in hexadecimal. What are the resulting pixel values? Are the pixels brightened?

(10%) b. [10] <2.8> Repeat part (a) but use saturating arithmetic instead. What are the resulting pixel values? Are the pixels brightened?
