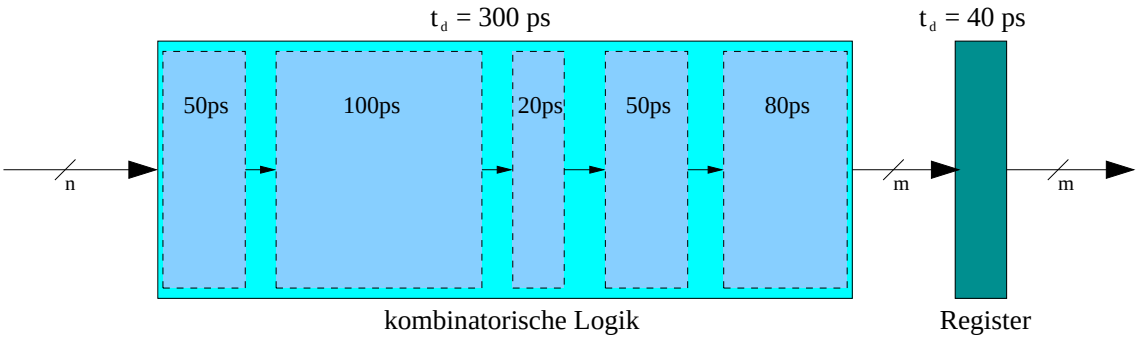
Aufgabenblatt 13 (Bonus)

Ausgabe 31/01/2011, Abgabe bis 07/02/2011 12:00

Name(n):
Matrikelnummer(n):
Übungsgruppe:

Aufgabe 13.1 Pipelining (5+5+5 Punkte)



Gegeben sei die oben gezeigte Funktionseinheit. Die Zeitangabe $t_d = 300\,\text{ps}$ sagt aus, dass die Reaktion auf einen Signalwechsel am Eingang des kombinatorischen Logikblockes nach $300\,\text{ps}$ am Ausgang erscheint. Für das Register soll der Einfachheit halber lediglich eine Zeitbedingung eingeführt werden. Die Angabe $t_d = 40\,\text{ps}$ soll hier bedeuten, dass das Register $40\,\text{ps}$ benötigt, um den Eingabewert zu verarbeiten.

a) Geben Sie die Latenzzeit („delay“) der obigen Schaltung sowie den zugehörigen Durchsatz an.

b) Der kombinatorische Logikblock der obigen Schaltung lässt sich, an den angegebenen Stellen in kleinere Logikblöcke aufteilen, wobei die Reihenfolge beizubehalten ist. Zeigen Sie, wie durch Einführung von Pipelineregistern (mit gleichen Zeitbedingungen wie das vorhandene Register) der Durchsatz maximiert werden kann.

Wir erlauben zunächst nur ein einziges Pipelineregister, also eine zweistufige Pipeline. An welcher Stelle muss das Pipelineregister für maximalen Durchsatz platziert werden? Geben Sie für diese Konfiguration Durchsatz und Verweildauer an.

c) Welche Werte ergeben sich für Latenz und Durchsatz, wenn an allen vier markierten Stellen Pipelineregister eingefügt werden?

Aufgabe 13.2 Speicherhierarchie (5+5+5 Punkte)

Wir betrachten das „Beispiel für ein einfaches Speichersystem“ im Skript (Foliensatz 10-14, Seiten 291ff oder Bryant/O’Hallaron 10.6). Überlegen Sie sich das Verhalten der angegebenen Speicherhierarchie mit Seitentabelle, Translation Lookaside Buffer und Cache. Vervollständigen Sie dann die Einträge auf den Folien „Adressumsetzungs-Beispiel 1-3“.

Aufgabe 13.3 Speicher-Gebirge („memory-mountain“) (10+20 Punkte)

Die Performance eines Rechners wird neben Taktfrequenz und Aufbau des Prozessors (also seiner Pipeline und Rechenwerken) vor allem von der Speicherhierarchie mit den Befehls- und Daten-Caches und dem Hauptspeicher geprägt. Im Lehrbruch von Bryant und O'Hallaron wird zur Analyse das C-Programm `mountain` beschrieben (siehe Abschnitt 6.6.1). Es misst die Ausführungszeit von Unterprogrammen abhängig von der Größe der beteiligten Daten. Dies lässt sich als Gebirge („memory mountain“) der Leserate („read-throughput“) in MByte/s als Funktion der Datengröße und des Abstands benachbarter Daten („stride“) darstellen.

a) Laden Sie die Quelldateien von der Webseite der Übung herunter (`mountain.tgz`) und compilieren Sie das Programm auf einer Maschine mit installierter GNU-Toolchain. Dazu sollte es ausreichen, einfach den Befehl `make` aufzurufen. Sie können das Programm auch von der offiziellen Download-Seite der CMU herunterladen:

http://csapp.cs.cmu.edu/public/code.html

Sie benötigen die Dateien clock.h, clock.c, fcyc2.h, fcyc2.c, mountain.c und ggf. ein Makefile. Alle benötigten Dateien finden sich auch unter:

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f05/code/mem/mountain/

b) Führen Sie das Programm — möglichst auf mehreren verschiedenen Rechnern — aus und speichern Sie die Ausgabe in Datei(en) ab (z.B. `./mountain > my-mountain.out`).

Was erhalten Sie als maximale Leserate in MByte/s Ihres Rechners, und welche Leserate bleibt davon für große Datenmengen noch übrig?

Versuchen Sie, die Daten als Gebirge zu plotten (z.B. Import in Excel oder Openoffice Calc, dann via 3D-Diagramm-Assistent).

Schauen Sie sich den Quellcode von `clock.c` an: wie wird dort die Ausführungszeit einer Funktion gemessen? Auf x86-Rechnern wird z.B. mit dem Assembler-Befehl `rdtsc` direkt der Timestamp-Counter ausgelesen (vergleiche Aufgabe 2.1).

Aufgabe 13.3 Buffer-Overflow (40 Punkte)

Achtung: die Aufgabe ist lehrreich, aber schwierig. Siehe auch Aufgabe 3.38 im Bryant & O'Hallaron und die Hinweise am Ende dieser Aufgabe. Laden Sie die Quelldatei `bufbomb.c` von der CS.APP Webseite herunter:

http://csapp.cs.cmu.edu/public/code.html

Die Funktion getbuf() ermöglicht durch die statische Dimensionierung des Arrays `buf` einen Buffer-Overflow. Compilieren Sie die Datei und versuchen Sie, einen Eingabestring zu finden, der den Stack kompromittiert und das Programm zur Ausgabe des Wertes -559038737 (bzw. `0xdeadbeef`) bringt:

```
prompt> ./bufbomb
Type Hex string:  30 31 32 33
getbuf returned 0x1
```

Die Eingaben erfolgen als Hex-kodierte ASCII-Zeichen. Auf diese Art ist es möglich, neben „normalen“ Texten (im obigen Beispiel '0' '1' '2' '3') auch beliebige Werte (Adresse, Integer, x86-Opcodes) an das Programm zu übergeben.

Hinweis: die möglichen Eingabe-Strings hängen natürlich vom verwendeten Betriebssystem und den Tools ab. Schicken Sie zum Beispiel ein Protokoll des disassemblierten Codes (objdump -d) und den relevanten Ausschnitt des Stacks vor und nach Kompromittierung durch den Buffer-Overflow mit ein.

Hinweis 2: Versuchen Sie, die Programmausführung schrittweise in einem Debugger zu beobachten. Ermitteln Sie die Lage von Stackpointer und Basepointer usw.

Hinweis 3: das manuelle Erstellen von Opcodes ist sehr aufwendig. Verwenden Sie ggf. einen Assembler (gcc, gas), um benötigte numerische Werte zu assemblieren und dann objdump -d um daraus die Bytewerte zu ermitteln, die Sie als String an bufbomb übergeben können.

Hinweis 4: moderne Betriebssysteme verwenden diverse Techniken, um das Ausnutzen von Buffer-Overflows zu erschweren: ggf. wird Programmcode auf dem Stack erkannt und geblockt („non-executable protection“) und bei mehrfachem Aufruf des Programms könnte der Stack an unterschiedlichen Adressen liegen („address space randomization“). Als Abhilfe für das letzte Problem den Debugger nicht verlassen, sondern im gdb das Programm immer wieder neu ausführen. Die „non-executable protection“ muss unter Linux ggf. bereits beim Booten abgeschaltet werden.