Aufgabe 1

Eine Schaltung "statem" zur Steuerung eines Datenpfads hat die beiden Ausgänge "shift_o" und "load_o" und den Eingang "in_i".

```
1 architecture beh of tb is
2   signal clk, reset, ins, load, shift : std_ulogic;
3 begin
4
5 --
6 -- hier werden clk, reset und ins generiert
7 --
8
9 statem_i0 : statem
10 port map (
11   clk     => clk,
12   reset   => reset,
13   in_i    => ins,
14   load_o  => load,
15   shift_o => shift
16 );
17 end architecture;
```

Die Schaltung "statem" soll verifiziert werden. In Abbildung 1 ist das Zustandsdiagramm abgebildet, nach dem sich die Schaltung "statem" verhalten soll. Die beiden Ausgänge sind als "load,shift" im Diagramm dargestellt, d.h. "1,0" bedeutet, dass load = "1" ist und shift = "0" ist.

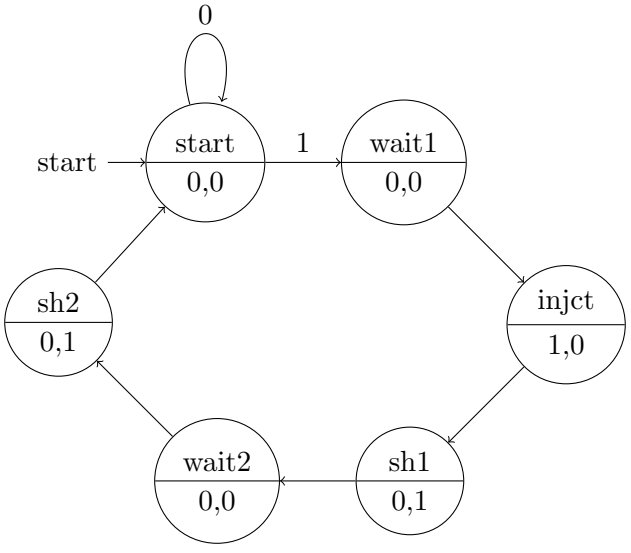

Abbildung 1: Zustandsdiagramm zur Spezifikation des Verhaltens der Schaltung "statem"

a) Sie wissen nichts über die Implementierung des Automaten. Wie viele Zyklen benötigen Sie um nachzuweisen, dass die Implementierung sich genau gemäß des Zustandsdiagramms in Abbildung 1 verhält?

b) Formulieren Sie eine assertion bezüglich der Signale "load" und "shift", die in jedem Takt gilt.

c) Wie viele Simulationszyklen benötigen Sie mindestens um jeden Zustand mindestens einmal erreicht zu haben und in jedem Zustand alle möglichen Eingangskombinationen ausprobiert zu haben? Zählen Sie die Zyklen ab der Deaktivierung des Resetsignals. Geben Sie eine Sequenz für das Eingangssignal "in_i" an, mit dem Sie diese minimale Anzahl von Zyklen erreichen können.

d) Nehmen Sie an, die Schaltung wurde mit folgender Eingangssequenz erfolgreich verifiziert: in_i = "111111100000001111111", d.h. die Schaltung hat sich für diese Sequenz genau gemäß des Zustandsdiagramms verhalten. Geben Sie ein Zustandsdiagramm für einen Automaten an, der sich für die Eingangssequenz genau gleich wie ein Automat gemäß des Zustandsdiagramms in Abbildung 1 verhält, der jedoch in folgenden Zyklen ein anderes Verhalten zeigen kann. Geben Sie eine Eingangssequenz für in_i an, bei der sich der Automat anders verhält.

e) Nehmen Sie an, dass Sie für die Verifikation auch den Zustand der Schaltung beobachten können. Begründen Sie, warum Ihnen dies bei der Verifikation helfen kann.

Lösung

a) Wenn nichts über die Implementierung bekannt ist und der innere Zustand nicht beobachtet werden kann, dann kann man keine Grenze für die Anzahl der Zyklen angeben. Für jede mögliche Anzahl an Zyklen lässt sich immer ein Automat angeben, der sich für diese Anzahl Zyklen genau gleich wie der geforderte Automat verhält aber im nächsten Zyklus ein Ausgangssignal produziert, dass unterschiedlich zum spezifizierten Zustandsdiagramm ist.

b) Die Signale "load" und "shift" dürfen laut Zustandsdiagramm nicht gleichzeitig "1" sein. Die Assertion wäre also:

```
1 assert not(shift = '1' and load = '1')
2             report "statem: shift and load both active"
3             severity failure;
```

c) Der Automat bleibt im Zustand "Start", wenn das Eingangssignal "0" ist. Um den Startzustand zu verlassen muss deshalb "in_i" auf "1" gesetzt werden. Danach ist die Zustandsfolge unabhängig von "in_i". Um jede Eingangskombination in jedem Zustand auszuprobieren muss der Graph zweimal durchlaufen werden, wobei im zweiten Durchlauf der Zustand von "in_i" invers zum Zustand beim ersten Durchlauf sein muss. Eine gültige Sequenz für "in_i" wäre beispielsweise "0111111100000". Es sind also 13 Zyklen notwendig.

d) In Abbildung 2 ist das Zustandsdiagramm eines Automaten dargestellt, der sich für die Eingangssequenz genau gleich wie ein Automat gemäß des Zustandsdiagramms in Abbildung 1 verhält, jedoch im nächsten Zyklus bei einer "0" am Eingang in einen Fehlerzustand geht.

e) Wenn bei der Verifikation gemäß der Methode nach Aufgabenstellung c) auch der Zustand beobachtet und überprüft wird, dann ist die Schaltung vollständig verifiziert. Damit wird genau die Situation aus Aufgabenstellung d) ausgeschlossen. Selbst die genaue Kenntnis der Zustandskodierung ist nicht notwendig. Bei einem Automaten gemaß Abbildung 2 gibt es einen Hinweis auf ein Problem schon durch den Unterschied von 0start und 1start. Allerdings könnte es dann noch sein, dass der Automat redundante Zustände enthält, die sich jedoch nicht durch unterschiedliche Ausgangssignale unterscheiden.

Angenommen vom Zustand "1start" würde bei einer "1" nach "0wait1" zurückgegangen, dann würde sich der Automat genau gemäß der Spezifikation in Abbildung 1 verhalten, jedoch wäre der Zustand "1start" redundant.

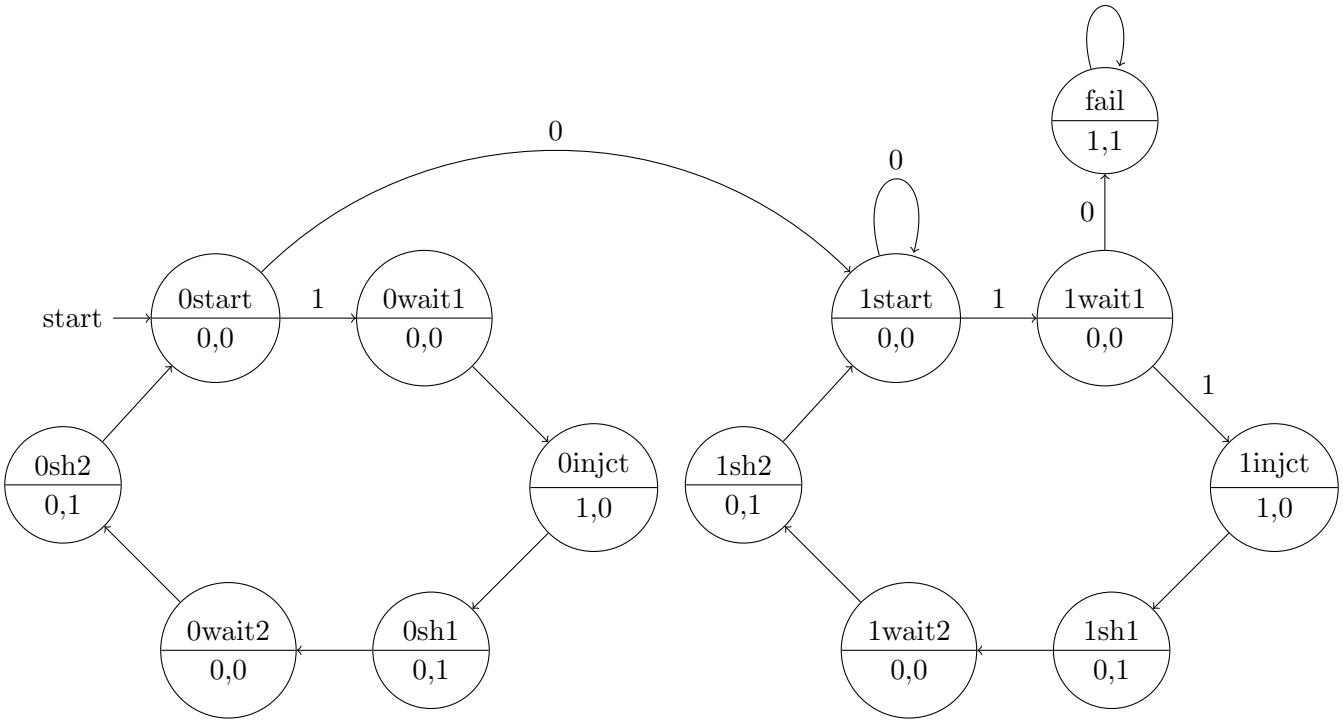

Abbildung 2: Zustandsdiagramm für den Automaten, der sich erst nach der Sequenz anders verhält

Aufgabe 2

Die Funktion room_id berechnet eine Identifikationsnummer für einen Raum aus den Elementen room_number, floor und building, die es für jeden Raum gibt. Die Daten der Räume sind in einem Array von structs gespeichert. Dieses Array hat 1000 Elemente, d.h. es gibt 1000 Räume. Nehmen Sie an, dass die Elemente des Arrays direkt nacheinander abgespeichert werden.

```
1 struct room_s {
2   int room_number;
3   int floor;
4   int building;
5 } rooms[1000];
6
7 int room_id(int room_index)
8 {
9 struct room_s *rp;
10
11 /* Compute the pointer to the room of index room_index */
12 rp = &rooms[0] + room_index;
13
14 /* Compute a room id number from room_number, floor and building number */
15 return (rp->room_number*1024 + rp->floor*16 + rp->building);
16 }
```

Nehmen Sie an Ihre Rechnerarchitektur hat einen Datencache mit einer Blockgröße von 16 Bytes und nehmen Sie an, dass das Array Cache Aligned ist, d.h. der Anfang des Arrays beginnt am Anfang eines Blocks. Nehmen Sie an, dass der Cache zu Beginn leer ist. Der Cache ist voll assoziativ und hat

eine Größe von 1 kByte. Im Cache sind nur die Arraydaten.

a) Wie hoch ist die durchschnittliche Cache-Miss Rate in Bezug auf den Zugriff auf Elemente der Structs? Die Funktion room_id wird 10 mal mit unterschiedlichen Parametern aufgerufen. Die Parameterwerte sind gleichverteilt, d.h. die Wahrscheinlichkeit, dass room_index 87 ist, ist genausogroß wie die Wahrscheinlichkeit für irgendeinen anderen Wert. Nehmen Sie an, dass der Cache zu Beginn der 10 Zugriffe leer ist.

b) Nehmen Sie an die durchschnittliche Cache-Miss Rate des Datencache beträgt 50%. In Ihrem Algorithmus sind 20% aller Instruktionen load Instruktionen. Wenn die Daten im Cache vorhanden sind, sind 2 Zyklen für den Zugriff notwendig. Wenn die Daten nicht im Cache sind, sind 40 Zyklen notwendig. Alle anderen Instruktionen benötigen 1 Zyklus. Berechnen Sie die durchschnittliche Anzahl der Zyklen pro Instruktion.

c) Nehmen Sie jetzt an, dass schon so viele Funktionsaufrufe von room_id() erfolgt sind, dass der Cache komplett gefüllt ist. Wie hoch ist die durchschnittliche Cache Miss Rate bei weiteren Aufrufen von room_id()?

d) Schlagen eine Änderung des structs room_s vor um die Cache Miss Rate zu minimieren. Wie groß ist die Cache Miss Rate mit Ihrem Vorschlag?

Lösung

a) In einem Struct sind 3 Integerwerte. Die Größe eines structs ist deshalb 12 Bytes. Die Blockgröße beträgt 16 Bytes. Deshalb ist nicht jedes Arrayelement in einem Block. Die Elemente 0, 4, 8, 12 ... und die Elemente 3, 7, 11 befinden sich vollständig innerhalb eines Blocks. Bei den Elementen 1, 5, 9, ... befindet sich das Strukturelement room_number in einem Block und floor und building in einem anderen Block. Bei den Elementen 2, 6, 10, ... befinden sich room_number und floor in einem Block, aber building ist im nächsten Block.

Die Wahrscheinlichkeit ist für alle 4 Varianten gleich. Wenn Arrayelemente 0,4,8,... oder 3,7,11,... geladen werden, dann beträgt die Cache Miss Rate 33%, da der Zugriff auf "room_number" zuerst erfolgt und mit dem Laden des 16 Byte Blocks in den Cache auch "floor" und "building" in den Cache geladen werden. Bei den Arrayelemente 1,5,9,... und 2,6,10,... müssen jeweils zwei Blöcke geladen werden. Die Missrate beträgt deshalb für diese Zugriffe 66%.

Mit jedem Zugriff steigt die Wahrscheinlichkeit, dass Daten schon im Cache vorhanden sind. Allerdings ist nach 9 Zugriffen die Wahrscheinlichkeit, dass ein Arrayelement wenigstens teilweise schon vorhanden ist bei 9/750 also etwa einem Prozent. In diesem Fall würde die Missrate von 33% auf 0% und von 66% auf 33% sinken. Durch diesen Effekt sinkt die Missrate um ca. 1%. (99 * 33% + 0%)/100 = 32%.

Ohne Berücksichtigung dieses Effektes liegt die Missrate bei (33% + 66%)/2 = 49%.

b) (80 Instruktionen * 1 Zyklus/Instruktion + 10 Instruktionen * 2 Zyklen/Instruktion + 10 Zyklen * 40 Zyklen/Instruktion)/100 Instruktionen = 500 Zyklen / 100 Instruktionen = 5 Zyklen pro Instruktion.

c) Der Cache hat eine Größe von 1kByte, d.h. es finden 64 Blöcke mit einer Größe von 16 Byte im Cache Platz. Die Hälfte der Arrayelemente liegt genau in einem Block und die andere Hälfte erstreckt sich über zwei Blöcke. Liegt ein Element komplett in einem Block sinkt die Cache Miss Rate von 33% auf 0%. Wenn ein Element in zwei Blöcken liegt, dann kann keiner der Blöcke, einer der Blöcke oder beide Blöcke im Cache liegen. Ist ein Block im Cache sinkt die Cache Miss Rate von 66% auf 33%. Wenn beide Blöcke im Cache liegen, dann sinkt sie auf 0%. Das gesamte Array ist 12000 Byte groß. Dies entspricht 750 Blöcken a 16 Byte.

Fall 1: Das Element liegt komplett in einem Block. Die Wahrscheinlichkeit, dass dieser Block schon im Cache liegt beträgt 64/750 = 8,5%.

Fall 2: Das Element liegt in zwei Blöcken. Die Wahrscheinlichkeit, dass beide Blöcke nicht im Cache liegen beträgt (750-64)/750 * (749-64)/749 = 83,7%. Die Wahrscheinlichkeit, dass beide Blöcke im Cache sind, beträgt 64/750 * 63/749 = 0,7%. Die Wahrscheinlichkeit für den Fall, dass einer im Cache und der andere nicht im Cache liegt beträgt etwa 15,8%.

In der Summe ergibt sich deshalb für die Cache Miss Rate (8,5% * 0% + 91,5% * 33% + 83,7% * 66% + 0,7% * 0% + 15,8% * 33%) / 200% = 45,3%.

Diese Berechnung berücksichtig nicht, dass die Blöcke im Cache nicht gleichverteilt ist. Durch die Struktur des Arrays und die Art des Zugriffs ist die Wahrscheinlichkeit, dass Block n im Cache ist höher, wenn Block n-1 schon im Cache ist. Es gibt also kleine Gruppen im Cache.

d) Wenn man ein zusätzliches Integerelement in den struct einfügt, dann ist jedes Arrayelement genauso groß wie ein Block im Cache. Die Gesamtgröße des Arrays steigt dadurch auf 16000 Bytes. Man hat aber bei jedem Aufruf der Funktion eine Cache Miss Rate von maximal 33%, da alle Daten eines Arrayelements in den Cache geladen werden, sobald auf ein Element des Structs zugegriffen wird. Wenn der Cache voll geladen ist, dann sinkt die Cache Miss Rate auf (1000-64)/1000 * 33% = 31%.