Programmieraufgabe 1

Aufgabe: Speichern und Rechnen

In einer heutigen Spielekonsole sind 8GB RAM (Arbeitsspeicher) verbaut.

a)

Als allererstes für diese Frage muss man sich bewusst machen, dass eine Binary Cell genau einen RS-Flip-Flop beinhaltet. Ein RS-Flip-Flop kann genau ein Bit speichern. Also hat kann auch eine Binary Cell genau 1 Bit speichern.

8 GB = 8 * 1024MB

1024MB = 1024 * 1024KB

1024KB = 1024 * 1024Byte

1024Byte = 1024 * 8 Bit

8 Bit = 8 * 1 Bit

→8 GB = 8 * 1024 * 1024 * 1024 * 8 * 1 Bit = 68719476736 Bit

→Man benötigt 68719476736 Binary Cells und somit 68719476736 RS-Flip-Flops.

Allerdings ist zu bemerken, dass heutzutage Konsolen einen dynamischen RAM besitzen, welches nicht mit einfachen Binary Cells möglich ist. Ich werde mich aber auch bei den folgenden Aufgaben auf einen statischen RAM beschränken.

b)

Ein Bussystem führt dazu, dass jede Zeile angesprochen wird. Jede Zeile hat sein eigenes Bussystem, welches durch einen Decoder angesprochen wird.

c)

Da Arbeitsspeicher wie eine Tabelle aufgebaut ist und wir in Aufgabe b) bereits die Wortbreite (64 Bit) bzw. die Spalten der Tabelle gegeben haben und in Aufgabe a) die Gesamtzahl der Binary Cells, können wir die Anzahl der Zeilen einfach ausrechen:

68719476736 / 64 = 1073741824 = 2^{30} → selbst der Decoder hätte 30 Eingänge.

Anhand dieser Unvorstellbar hohen Zahlen kann man erneut erkennen warum heutzutage im Arbeitsspeicher nicht mehr Binary Cells sondern mit Transistoren und Kondensatoren gearbeitet wird. Über 1 Milliarde Leitungen fehlerfrei, kompakt und kostengünstig zu verlegen ist unmöglich.

Zum Rechenwerk:

a)

Ich definiere die erste Zahl als Zahl x und die zweite Zahl als Zahl y und werde diese in den nächsten beiden Aufgaben so bezeichnen. x und y sind beide 64 Bit groß.

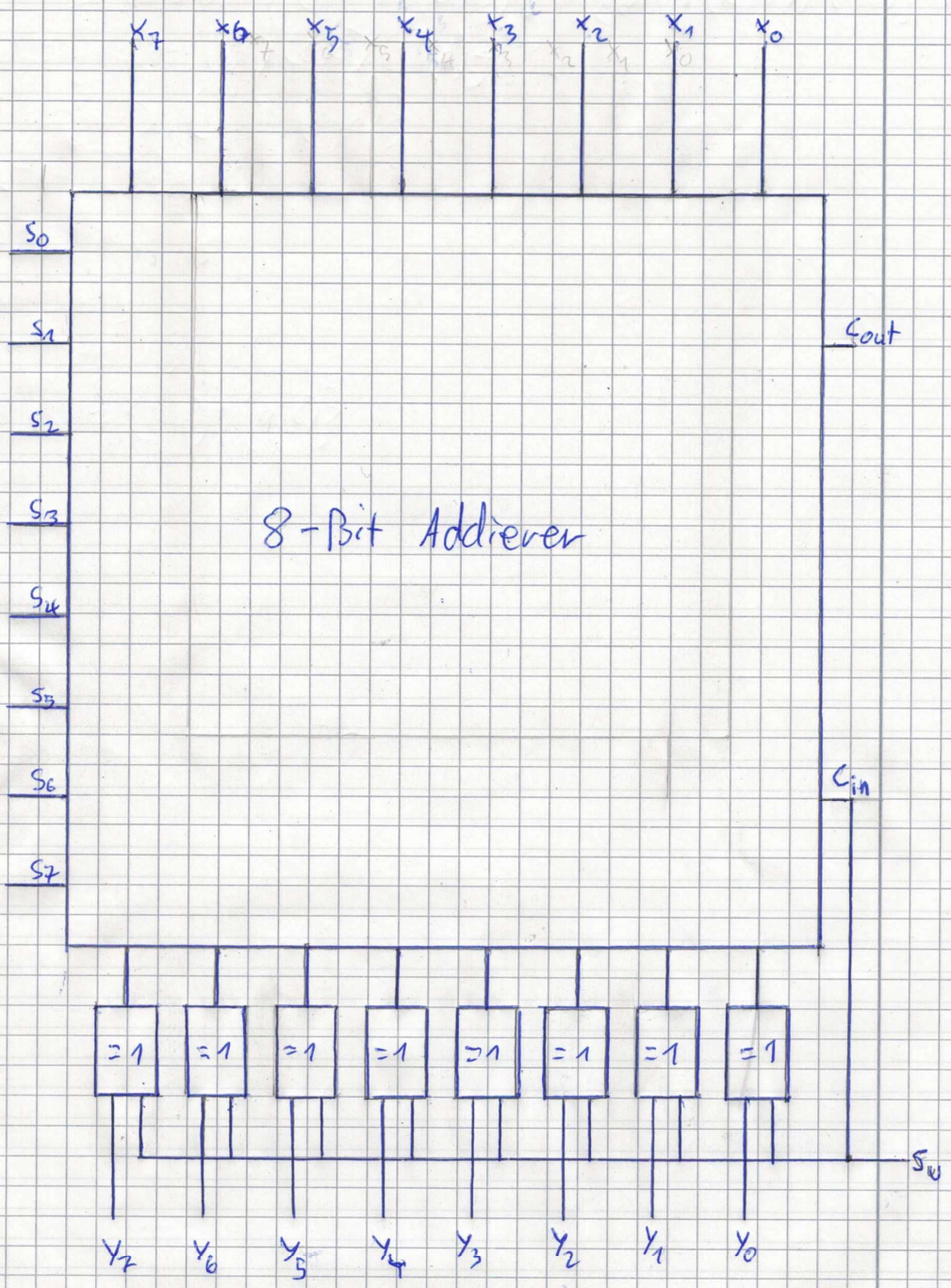
Wenn 2 Bitzahlen mit einer addiert werden, wird immer die erste Ziffer von x (x_0) mit der ersten Ziffer von y (y_0) verrechnet. Danach die zweite Ziffer von x (x_1) mit der zweiten Ziffer von y (y_1) usw.
Wenn x > 1 ist, dann braucht man dafür einen Volladdierer, denn nur dieser kann mit Überträgen umgehen. Jeder Volladdierer besteht aus 2 Halbaddierern.

Da wir 64 Bit Zahlen haben brauchen wir 64 Volladdierer und somit 128 Halbaddierer. In unserem Beispiel bräuchten wir allerdings nur 127, weil wir die bei x_0 und y_0 keinen Übertrag haben. In der Praxis sollte man jedoch mit Volladdierern auch beim ersten Bit arbeiten, denn es kann immer sein, dass es irgendwoher doch noch einen Übertrag gibt.

b)

1. Ein ALU besteht aus einem arithmetischen und einem logischen Teil für uns ist nur der arithmetische Teil wichtig.
2. Ein addiert mit Volladdierern(in unserem Fall 64-Bit Addierer) wie in Aufgabe a) des Rechenwerks.
3. Subtraktion funktioniert durch das Komplement.
4. Das Komplement ist die gleiche Zahl mit umgekehrten Vorzeichen.
5. Das Komplement wird gebildet in dem man all Bits der Zahl umdreht: 0 → 1 und 1 → 0 und danach 1 dazu addiert. Bsp.: 0010 → 1101 → 1110
6. Um zu kontrollieren ob wir Subtraktion oder Addition haben wollen brauchen wir eine Steuerleitung (Su).
7. Wenn wir y von x subtrahieren wollen müssen wir bevor y mit dem 64-Bit Addierer verbinden das Komplement bilden.
8. Dies funktioniert in dem wir jedes einzelne Bit mit der Steuerleitung durch ein XOR-Gatter verbinden. Wenn bei der Steuerleitung eine 0 ankommt wird dadurch addiert und wenn eine 1 ankommt werden die Bits invertiert und das Komplement gebildet.
9. Allerdings muss zum gebildeten Komplement noch 1 dazu addiert werden. Dazu verbinden wir die Steuerleitung mit dem ersten Übertrag (C_{in}) des ersten Volladdierers(x_0+y_0).
10. Für Subtraktion braucht man somit auf jeden Fall die 128 Halbaddierer bei einem 64-Bit Addierer.

Beispiel: 8-Bit Addierer



XOR

B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Aufgabe: Optimierung einer Schaltung

Programmieren Aufgabe: Optimierung einer Schaltung

a)

d	c	b	a	y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

b)

$$y = (\neg a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (a \wedge \neg b \wedge c \wedge \neg d) \vee (\neg a \wedge b \wedge c \wedge d) \vee (a \wedge b \wedge c \wedge d)$$

c)