Gruppe: David Greiner, Steffen Peikert Blatt 1 Semester: 6

Aufgabe 1:

1.1: AND, OR, NOT:

Als erstes schauten wir uns noch einmal die Vorlesungsfolien zur Modellierung von Entities und Architectures an. Dabei ist uns aufgefallen, dass für AND und OR jeweils die gleiche Entity verwendet werden konnte und nur eine andere Architecture benötigt wurde. Allerdings hätten wir dazu die Implementierung entweder in eine gemeinsame Datei schreiben, oder die Entities von den Architectures trennen müssen.

In einem Ersten Versuch erstellten wir deshalb eine einzige Datei (was sich später bei der Simulation als Problem herausstellte).

Da bei jedem der angegebenen Gatter immer ein Spezialfall existierte, liesen sich die einzelnen Logik-Bausteine durch jeweils eine Bedingung realisieren (oder durch das Einbinden der schon existierenden std_logic-Bibliothek und der Verwendung der vordefinierten Operatoren and, or oder not).

1.2: NAND, NOR:

Wie schon bei 1.1, konnten wir hier wieder dieselbe Entity wie für AND und OR verwenden. Gleichzeitig lassen sich die beiden geforderten Gates durch jeweils ein AND+NOT und OR+NOT realisieren, weshalb wir hier auf die schon bestehenden Gatter zurückgegriffen haben. Hierbei bereitete uns vor allem die Syntax ein großes Problem, da wir nicht direkt aus den Folien erkennen konnten, wie genau eine solche zusammengesetzte Architektur erstellt wurde. Erst mithilfe einiger Beiträge aus dem Internet und der Nachfrage bei Herrn Bromberger gelang es uns, die Blöcke zu implementieren.

1.3: Impuls:

Bei dieser Aufgabe lag das Problem im Herausfinden, wie genau Variablen und Konstanten definiert und entsprechend verwendet werden. Die Vorlesungsfolien und ein Beispiel in einem Forum reichten aus, um jegliche Unklarheiten zu beseitigen. Wie auch schon bei 1.2 und 1.1 lies sich hier wieder eine der vordefinierten Entities wiederverwenden.

Da in der Aufgabenstellung stand, das ein Taktsignal an den Block gesendet wurde, aber dort nur Taktzyklen gezählt werden sollen, verwendeten wir eine rising_edge Bedingung (alternativ falling_edge), um nur bei einer Änderung des Taktes den Counter zu erhöhen.

1.4: Counter:

Hier lies sich die Implementierung unseres Counters fast 1:1 wieder verwenden. Die einzigen Änderungen waren, dass nun noch eine weiter Variable hinzukam, die den aktuellen Ausgangszustand enthielt. Diese wurde dann einfach nach entsprechend vielen Zyklen invertiert.

Aufgabe 2:

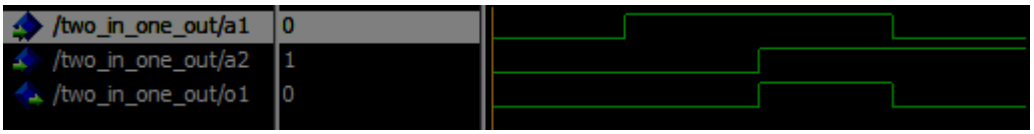
2.1.1: Modelsim

Hier testeten wir unsere Schaltungen. Mithilfe der Vorlesungsfolien lies sich sehr einfach ein Projekt einrichten. Allerdings hatten wir einige syntaktische Fehler, die wir nun mittels Trial&Error einzeln beheben konnten.

Ein weiteres Problem war durch das Verwenden einer Entity für alles entstanden. Wir konnten nicht das ganze Projekt simulieren, sondern mussten immer eine einzelne Architecture auswählen und testen. Deswegen (und weil es die Aufgabe gefordert hatte) trennten wir unsere Implementierung auf.

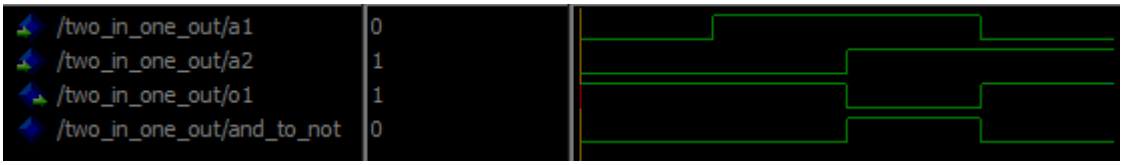
Im Folgenden sind die Ergebnisse, die wir bei der Simulation bekommen haben, sie zeigen dass die Gates wie erwartet funktionieren.

AND:

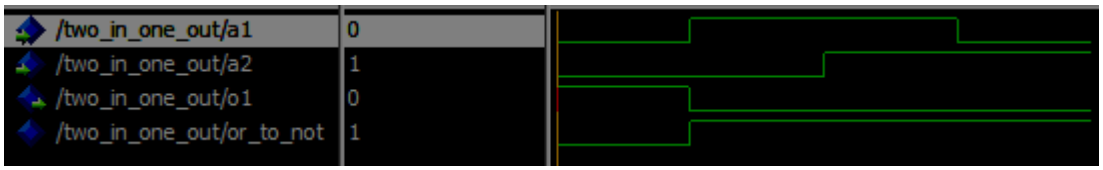


OR:

NAND:



NOR:



Impuls:

Counter:

2.1.2: Testbench

Zu Beginn hatten wir die Testbenches in extra Dateien ausgelagert, nach dem das Probleme gemacht hat haben wir sie in unsere existierenden Dateien integriert. Da haben sie nach einigem ausprobieren dann funktioniert. Nachdem die erste funktioniert hatte war der Rest nach Schema f sehr leicht zu erledigen.
Dabei war es ziemlich „straight forward“ mit Signalen und asserts die jeweiligen Bausteine zu testen.

2.1.3: Scripts

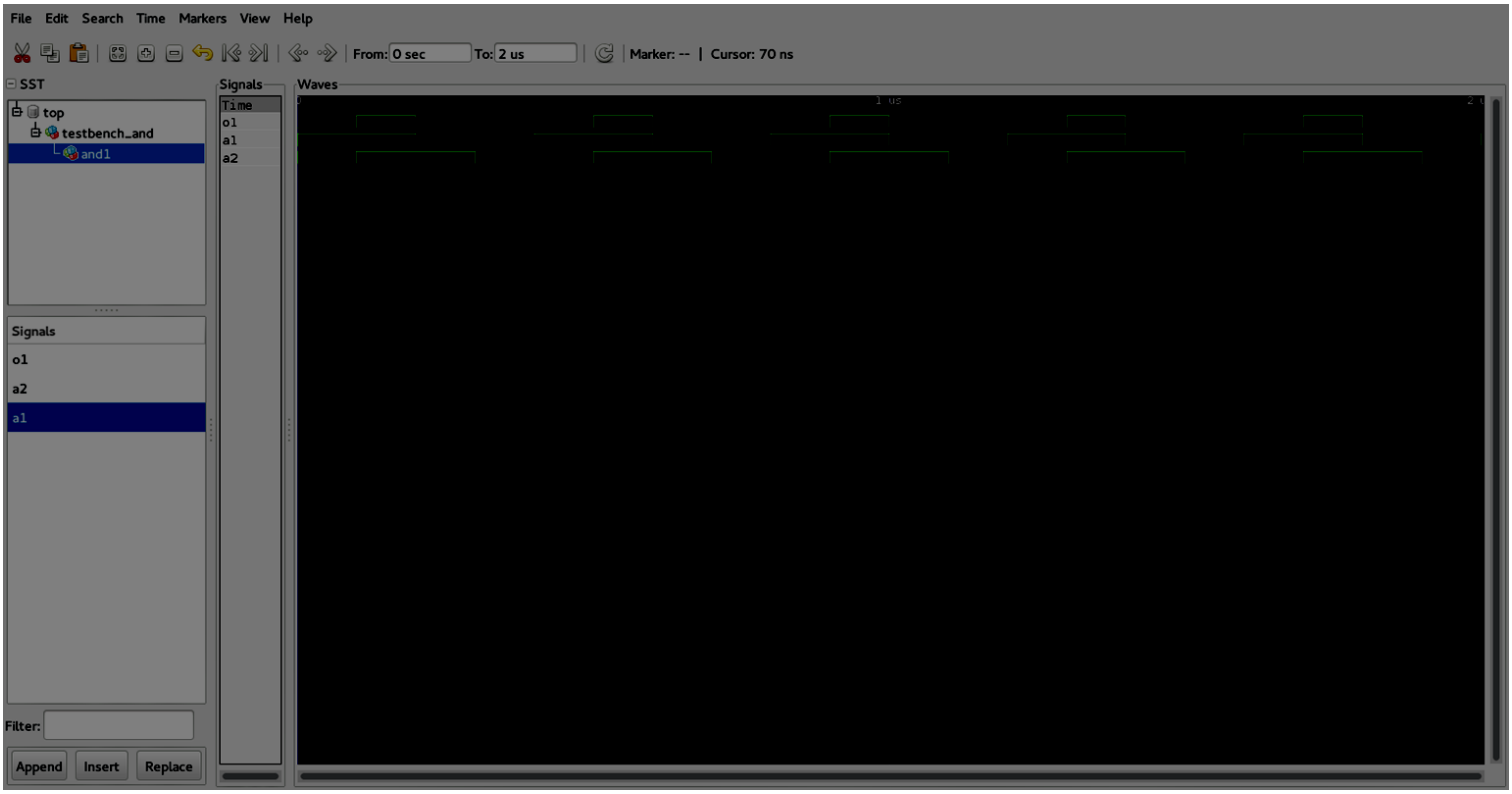
Zunächst haben wir die Befehle anhand ausprobieren in Modelsim ausgelesen, da jedoch nicht alles gut erklärt ist haben wir im Internet eine sehr gute Referenz gefunden: http://cseweb.ucsd.edu/classes/fa10/cse140L/lab/docs/modelsim_ref.pdf
Die uns geholfen hat die Skripte zu ergänzen.

2.2.1: GTKWave

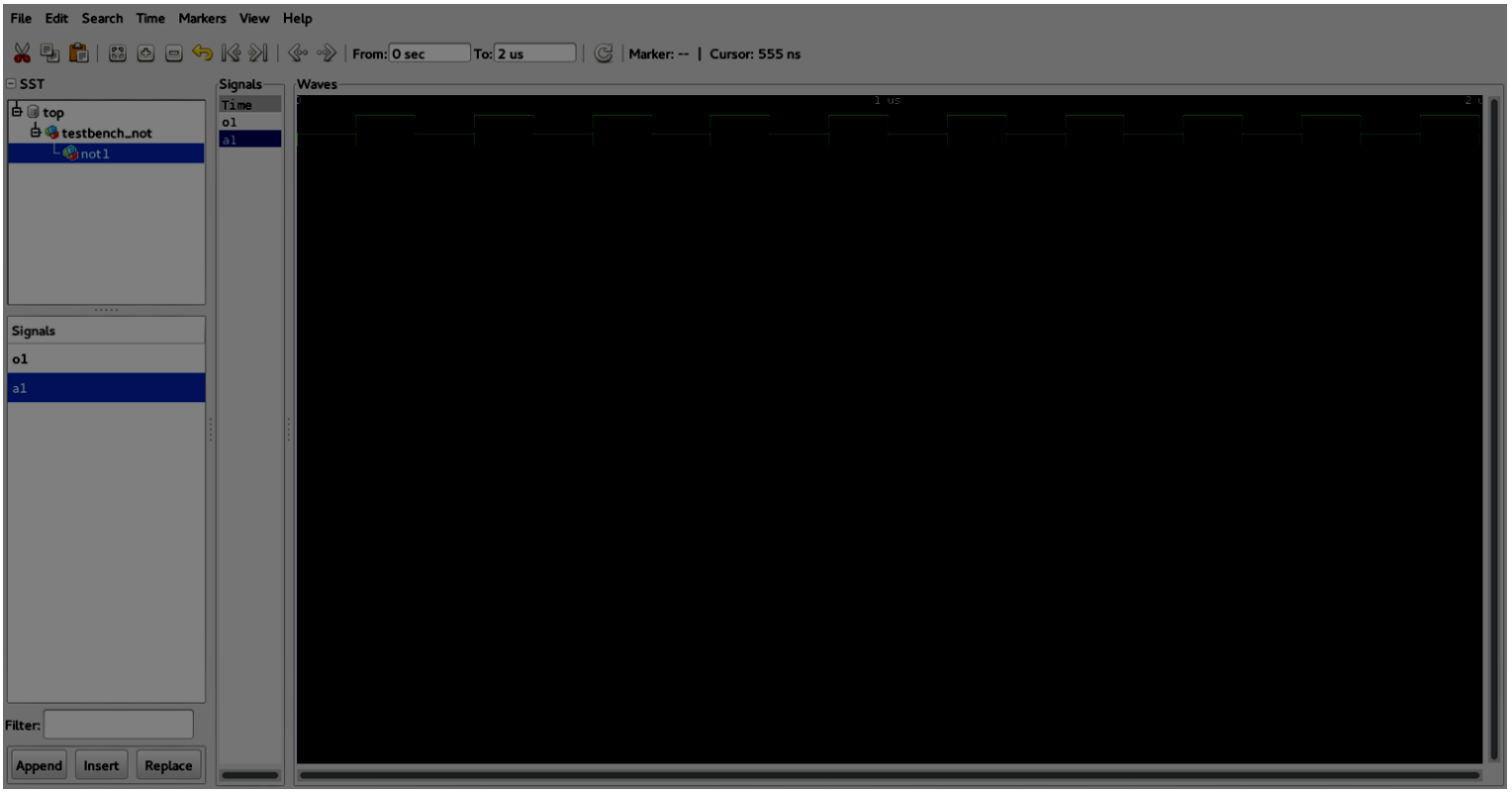
Auch mit Hilfe der ghdl konnten wir zeigen dass unsere Schaltungen die richtigen Ergebnisse liefern, dabei war die Bedienung über die Kommandozeile zunächst ganz im Kontrast zu dem vorher benutzten Modelsim, jedoch nach kurzer einarbeiten sehr übersichtlich und klar verständlich.

Gruppe: David Greiner, Steffen Peikert Blatt 1 Semester: 6

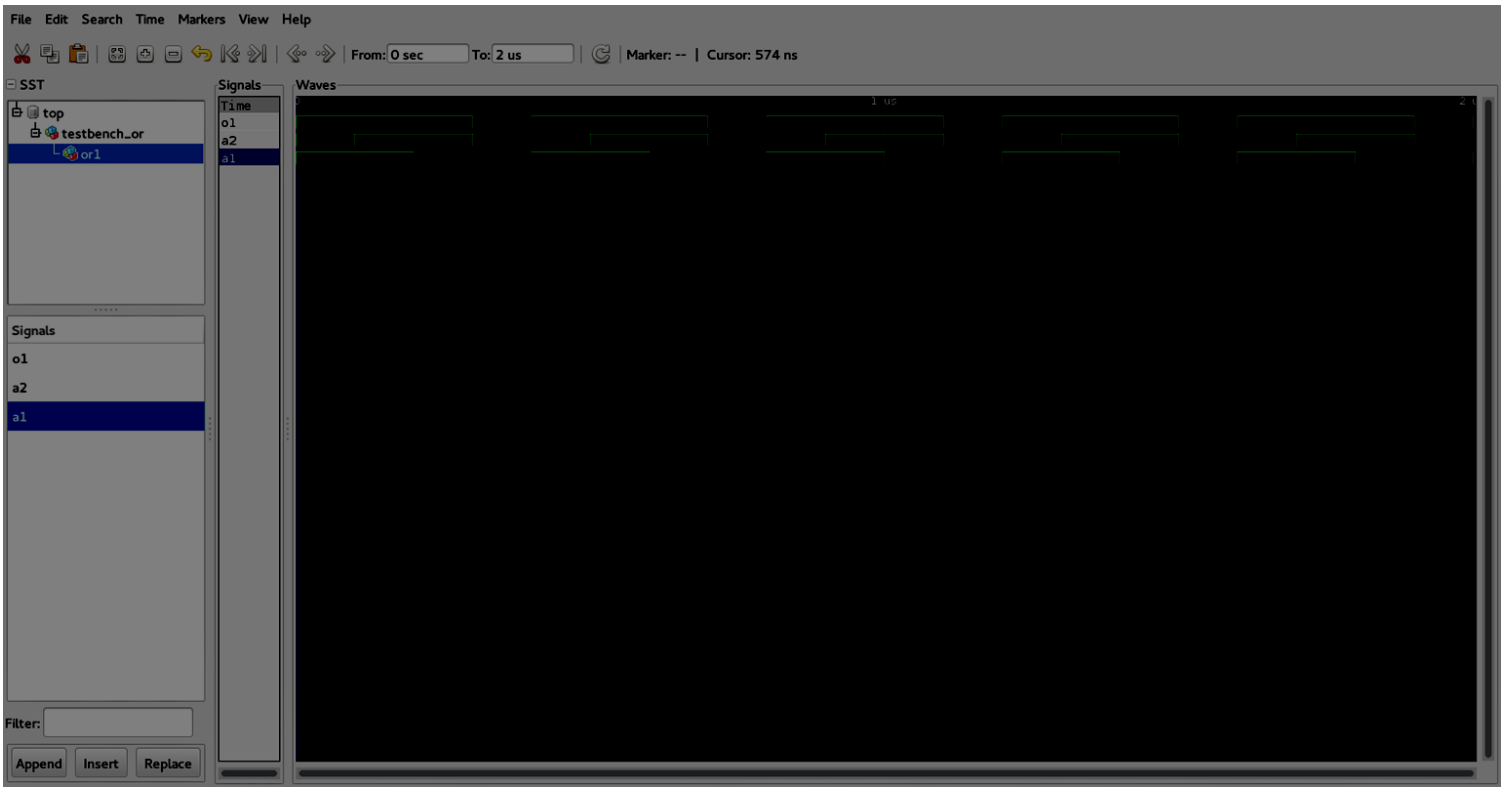
AND:



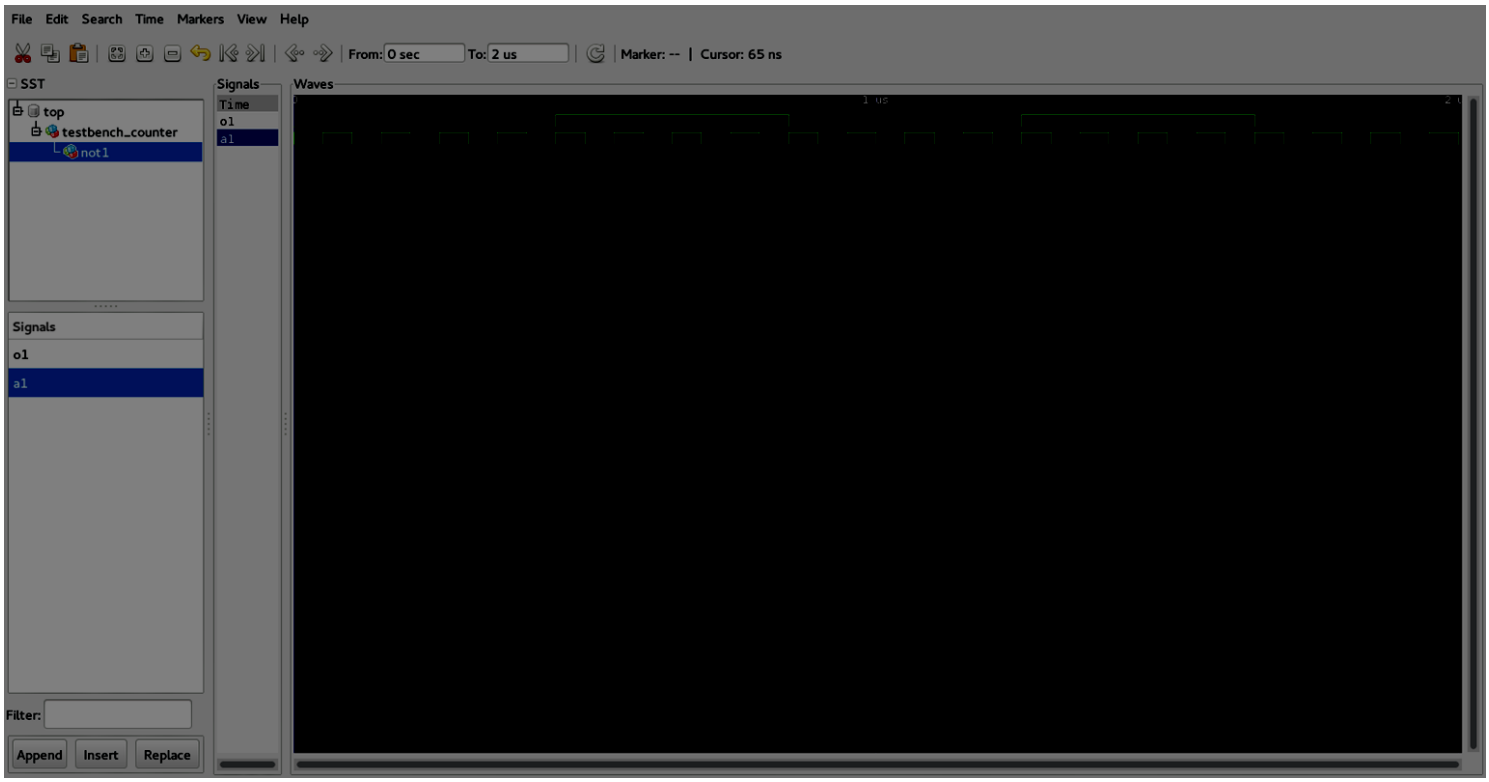
NOT:



OR:


File Edit Search Time Markers View Help
From: 0 sec
To: 2 us
Marker: -- | Cursor: 574 ns
SST
top
testbench_or
or1
Signals
o1
a2
a1
Filter:
Append
Insert
Replace
Waves
Time
1 us

COUNTER:


File Edit Search Time Markers View Help
From: 0 sec
To: 2 us
Marker: -- | Cursor: 65 ns
SST
top
testbench_counter
not1
Signals
o1
a1
Filter:
Append
Insert
Replace
Waves
Time
1 us

2.3: Auswertung

	Modelsim	GHDL
Vorteile	– Graphische Oberfläche – Direkte Projektübersicht – Kompilierreihenfolge im Tool änderbar – Sowohl Editor als auch Simulations- & Wave-Tool in einem keine „begrenzte“ Simulation (kann um x schritte erweitert werden)	– Minimalistisch – Direktere Fehlermeldung (besser Verständlich) – Freie Software – Nativ auf Linux verfügbar
Nachteile	– Fehlermeldung schwer zu finden (doppelklick auf roten text?!) – Fehlermeldungen kryptisch – Unübersichtliche Menüstruktur – kostenpflichtig – nicht nativ auf Linux verfügbar (auch nicht die Studentenversion)	– Terminal/Konsole (wobei man hier auch für PRO argumentieren kann, ich (Steffen) bin ein Freund von Konsolen) – Sekundärtool (gtkwave) notwendig, um simulierte wave zu sehen – Vordefinierte Anzahl an simulierten Sekunden

Auch wenn wir zu beginn mit Modelsim gearbeitet haben, wollen wir eher mit GHDL/GTKWave weiter arbeiten, da wir meistens unterwegs mit dem Laptop (Linux) arbeiten. Zudem sind uns die genaueren Fehlerbeschreibungen wichtiger als eine schön aussehende GUI.
Trotzdem kann ich mir vorstellen, dass sich Modelsim besser für größere Projekte mit mehreren Dateien eignet als GHDL.