Zusammenfassung Architektur Eingebetteter System

Paul Nykiel

27. Juli 2019

This page is intentionally left blank.

Inhaltsverzeichnis

Kapitel 1

Einführung

Ein eingettetes System ist in einen technischen Kontext oder Prozess eingebettet.
Im wesentlichen kann ein eingebettetes System als ein Computer, der einen technischen Prozess steuert oder regelt, betrachtet werden. Grafik

1.1 Architektur eines Eingebetteten Systems

1.1.1 Eigenschaften eines Eingebetteten Systems

- Enge Verzahnung zwischen Hard- und Software
- Strenge funktionale und zeitliche Randbedinungen
- Zusätzlich zum Prozessor wird I/O Hard- und Software benötigt
- Oftmals wird Anwendungsspezifische Hardware benötigt

⇒ Keine „General-Purpose“ Lösung möglich
Zusätzliche Probleme:

- Wenig Platz
- Nur beschränkte Energiekapazität
- System darf nicht warm werden
- Kostengünstig

1.1.2 Zusätzliche Herausforderungen beim Entwurf

Die Entwicklung eines eingebetteten Systems ist kein reines Software-Problem, zusätzlich muss beachtet werden:

- Auswahl eines Prozessors, Signalprozessors, Microcontrollers

- Ein-/Ausgabe Konzept&Komponenten
  - Sensoren und Aktoren
  - Kommunikationsschnittstellen
- Speichertechnologien und Anbindung
- Systempartitionierung: Aufteilen der Funktionen der Komponente
- Logik- und Schaltungsentwurf
- Auswahl geeigneter Halbleitertechnologien
- Entwicklung von Treibersoftware
- Wahl eines Laufzeits-/Betriebssystems
- Die eigentliche Softwareentwicklung

⇒ Aufteilung des Entwurfs auf mehrer Entwurfsebene

1.1.3 Entwurfsebenen

Verhalten	Syntheseschritt	Entscheidungen	Test
System Specification	Systemsynthese	HW/SW/OS	Modelsimulator / Checker
Behavioural Specification	Verhalten / Architektursynthese	Verarbeitungseinheiten	HW/SW-Simulation
Register-Transfer-Specification	RT-Synthese	Register, Addierer, Mux	HDL-Simulation
Logic-Specification	Logiksynthese	Gatter	Gate-Simulation

Tabelle 1.1: Entwurfsebenen

Grafik

1.2 Hardwarespezifikationssprachen

- Verilog
- VHDL (Very High Speed Integrated Circuit Description Language)

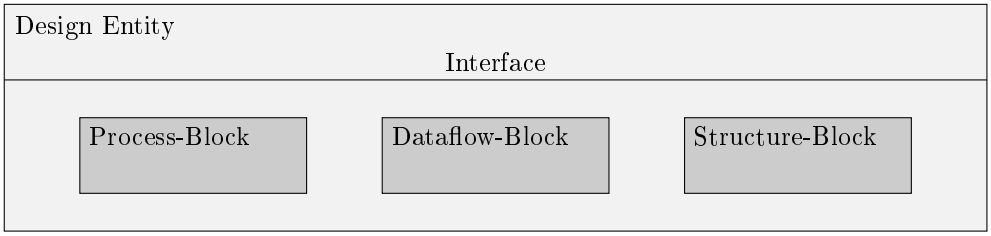

Abbildung 1.1: Aufbau einer Design-Entity

Process-Block Sequentiell abgearbeitete Logik:

```
process(clk)
begin
    ...
end
```

Dataflow-Block Konkurrent abgearbeitete Logik:

```
begin
    ...
end
```

Structure-Block Zusammenschalten weiterer Design-Entitys:

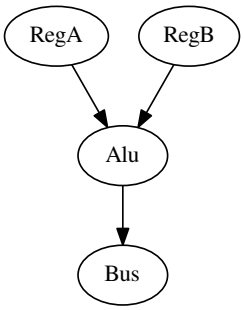

Abbildung 1.2: Structure-Block

1.2.1 Aufbau von VHDL-Beschreibungen

- use: Import von Bibliotheken

- `entity`: Schnittstellenbeschreibung
- `architecture`: Implementierung der Entity
- `configuration`: `architecture` zu `entity` auswählen

1.2.2 Beispiel: Multiplexer

Entity-Deklaration:

```
entity MUX is
    port(a,b,sel: in Bit;
         f: out Bit);
end MUX;
```

Als Process-Block

```
architecture BEHAVIOUR_MUX of MUX is
begin
    process(a,b,sel)
    begin
        if sel = '1' then f <= a;
        else f <= b;
    end process;
end BEHAVIOUR_MUX;
```

Als Dataflow-Block

```
architecture DATAFLOW_MUX of MUX is
begin
    f <= a when sel = '1' else b;
end DATAFLOW_MUX;
```

alternativ geht auch:

```
architecture DATAFLOW_MUX of MUX is
begin
    f <= (a and sel) or (b and (not sel));
end DATAFLOW_MUX;
```

eine weitere Option:

```
architecture DATAFLOW_MUX of MUX is
signal nsel, f1, f2 : Bit;
begin
    nsel <= not sel;
    f1 <= a and sel;
    f2 <= b and nsel;
    f <= f1 or f2;
end DATAFLOW_MUX;
```

Alternativ: Mit Variablen

Laut Skript geht das so nicht, sollte aber eigentlich schon?

Als Structure-Block

```
architecture STRUCTURE of MUX is
    component NOT
        port(i: in Bit; o: out Bit);
    end component;
    component AND
        port(i1, i2: in Bit; o: out Bit);
    end component;
    component OR
        port(i1, i2: in Bit; o: out Bit);
    end component;
    signal nsel, f1, f2: Bit;
begin
    g1: AND port map(a, sel, f1);
    g2: AND port map(b, nsel, f2);
    g3: OR port map(f1, f2, f);
    g4: NOT port map(sel, nsel);
end STRUCTURE;
```

1.3 Configuration

Rekursive die Architektur für jede Entity auswählen:

```
configuration STRUCTURE_MUX of MUX is
    for STRUCURE
        for g1,g2: AND use entity MYAND(BEHAVIOUR_AND)
            ...
        end for;
    end for;
end STRUCTURE_MUX;
```

Was genau passiert da jetzt?

Dann muss die oben genutze Entity und Architektur natürlich noch definiert werden:

```
entity MYAND is
    port(i1, i2: in Bit;
         o: out Bit);
end MYAND;

architecture BEHAVIOUR_MYAND is
    o <= i1 and i2;
end BEHAVIOUR_MYAND;
```

1.4 VHDL-Simulationssemantik

Aufgaben des Simulators:

- Signal treiben/propagieren
- Rückkopplungen auflösen
- Verzögerungen modellieren

1.4.1 Signale Treiben

Signale werden durch eine Event-Queue repräsentiert, das heißt nicht die Signale selber, sondern nur Änderungen der Signale (z.b. Flanken) werden gespeichert. Die Event-Queue besteht aus „Transaktionen“ jede Transaktion ist ein Tuple aus der Zeit zu der die Änderung auftritt, und der Änderung selber.
Zum Beispiel:

```
y <= '0' after 0ns, '1' after 10ns, '0' after 20n;
```

Wird als Event-Queue so dargestellt:

$$\{<0,'0'>,<10,'1'>,<20,'0'>\}$$

1.4.2 Rückkopplungen auflösen

Transaktionen können echt parallel stattfinden (Ereignisse treten asynchron und ggf. gleichzeitig auf). ⇒ Es kann zu Konflikten kommen („Henne-Ei-Problem“), z.B. bei einem zero-delay RS-Latch:

```
1  x <= not (y and lset);
2  y <= not (x and reset);
```

Lösung: Tagged-Event-Queue bzw. Delta-Delay: Jeder Zeitpunkt wird um eine „zweiten Dimension“ ergänzt, Events die direkt nacheinander (z.B. als direkte Folge) auftreten (mit einem Delta-Delay) werden entlang dieser zweiten Dimension geordnet.

t	lset	x	y	reset	Zeile
0	↓	0	1	1	1
$0+\Delta$	0	↑	1	1	2
$0+2\Delta$	0	1	↓	1	1
$0+3\Delta$	0	1	0	1	✓
10	↑	1	0	1	1
$10+\Delta$	1	1	0	1	✓

Tabelle 1.2: Tagged-Event-Queue für den zero-delay RS-Latch

1.4.3 Verzögerungen modellieren

Durch Schaltzeiten, Kapazitäten, Laufzeiten etc. kommt es in echten Systemen zu Verzögerungen der Signale. Diese müssen daher auch in VHDL modelliert werden können. Dafür wird zwischen zwei Arten unterschieden:

- Langsames Ansprechverhalten, das heißt kurze Pulse werden nicht durchgelassen
- Verzögerung der Signale

Beispiel (Inverter):

```
out <= reject 10 ns inertial not in after 30ns;
```

Die Verzögerungen werden in VHDL durch `reject` für langsames Ansprechverhalten (hier muss der Puls mindestens 10ns dauern) und `after` für Verzögerungen (hier 30ns) modelliert.
Für reine Laufzeitverzögerungen kann in VHDL auch `transport` genutzt werden, folgende Befehle sind äquivalent:

```
out <= transport in after 30ns;
out <= reject 0ns inertial in after 30ns;
```

wait_until und generic

Kapitel 2

Processing Elements

2.1 Instruction Set Processor (ISP)

2.1.1 von-Neumann-/Princeton-Architektur

Grafik

2.1.2 Befehlszyklus

1. Befehl holen (fetch)
2. Befehl dekodieren (decode)
3. Operanden holen (load)
4. Befehl ausführen (execute)
5. Daten speichern (write back)

Pro	Contra
Analyse einfach Speicher flexibel benutzbar	Auslastung gering von-Neumann Flaschenhals (Daten und Befehle über den selben Bus)

2.1.3 Harvard-Architektur

Grafik

Pro	Contra
Auslastung Kein Flaschenhals Schnellere Abarbeitung	Fragmentierter Speicher Analyse schwierig Schwierig bei Datenabhängigkeiten

2.2 Application Specific Instruction Set Processor (ASIP)

Regulärer ISP wird durch zusätzliche Instruktionen (und damit auch Hardware) für ein bestimmtes Einsatzgebiet optimiert, z.B. durch zusätzlich „Multiply-Accumulate“-Einheit oder auch komplette FFT-Operationen. Beispiel: DSP (Digital Signal Processpr). Grafik

2.3 Application Specific Processor

Nicht mehr programmierbar, Prozessor kann nur wenige vorkonfigurierte Befehle ausführen, Steuerung erfolgt über eine spezielle Logik, oftmals in Form einer State-Machine. Grafik

2.4 Beispiel: Aufbau eines Schnurlosen DECT-Telefons

Oftmals wird ein ISP mit zugehöriger RF-Hardware auf einem Chip-Integriert (z.B. in Mobiltelefonen), dann spricht man von einem Sytem-on-a-Chip (SoC). Grafik

2.5 Wie kommunizieren PEs in heterogenen Systemen?

2.5.1 Gemeinsame Ressourcen (Speicherkopplung)

Alle PEs können über einen Bus auf eine gemeinsame Ressource (für gewöhnlich Speicher) zugreifen. Für die Synchronisation ist ein „Arbiter“ (Richter) sowie ein Bus-Controller in jedem PE notwendig. Grafik

2.5.2 Direkte Verbindung

Direkte Verbindung zwischen allen PEs die kommunizieren müssen, Arbitrierung ist nicht notwendig, mehrere PEs können gleichzeitig kommunizieren. Es ist wieder ein Controller für jede Verbindung von nöten, zudem muss dieser eventuell Daten puffern. Bei n PEs sind im Worstcase $\frac{(n-1)\cdot n}{2} \in \mathcal{O}(n^2)$ Verbindung notwendig.