3.6 Ausnahmebehandlungen

Zur Vorlesung Rechenanlagen

SS 2019

Exceptions

Wir hatten schon beim Thema „Überlaufbehandlung“ auf spätere Kapitel verwiesen. Bisher haben wir nur die Möglickeit wie folgt auf Überläufe zu reagieren:

…

```
ADD R3,R5,R4;
BNOV #1;
JAL R2,#Überlaufbehandlung;
```

Bei rückkehrender Behandlung (R2 müsste man dann immer frei halten!), bzw. sonst

…

```
ADD R3,R5,R4;
BOV #Überlaufbehandlung;
```

Das wäre ein Riesenaufwand, wollte man das bei jeder Arithmetikoperation tun, die einen Überlauf erzeugen kann!

Exceptions

Bei einem Überlauf kann es auch sinnvoll sein das Programm fortzusetzen (rückkehrende Behandlung).

Es kann aber zu Situationen kommen, in denen ein Programm nicht mehr sinnvoll fortgesetzt werden kann:

Da wir beliebig in den Speicher schreiben dürfen, kann es zum Beispiel vorkommen, dass kein legaler Befehlscode mehr im Programmspeicher an der Stelle PC steht, weil

- wir den PC falsch gesetzt haben (z.B mit JREG)
- ein STORE auf eine Adresse im Programmbereich gemacht haben.

In diesen Fällen sollte die CPU trotzdem noch sinnvoll reagieren können. **Wie?**

Exceptions

Solche, vom Programm verursachten, Situationen sollte die Maschine durch ihre Kontrolle abfangen, um

- das Programm beenden oder
- nach Bereinigung der Situation wieder fortsetzen

zu können.

Lösungsidee:

Implementiere beim Vorliegen solcher Ausnahmesituationen einen „automatischen Unterprogrammaufruf“ (ggf. des Betriebssystems), der zur Bearbeitung des auslösenden Befehls zurückkehren kann.

Wir nennen solche **interne** Unterbrechungen **Exceptions** oder **Ausnahmebehandlungen.**

Interrupts

Wenn man mal den Schritt gemacht hat, dass die CPU auf interne, vom Programm verursachte Ereignisse reagiert, kann man diesen auch auf externe Ereignisse erweitern.

Mit unserer bisherigen Maschine müssten wir langsame Geräte (Platten, Netzwerk, Tastatur, Maus,…) in mehr oder weniger regelmäßigen Zeitabständen auf ihren Zustand prüfen (polling). Dazu müsste man alle Programme so schreiben, dass sie z.B. alle 50 Befehle äußere Ereignisse abfragen und diese via Unterprogramm behandeln.

Das wäre ein unvertretbarer Aufwand ähnlich wie bei Überläufen.

Da diese Unterbrechungen nicht vom Programm selbst, sondern von außen verursacht werden, bezeichnen wir sie als (externe) **Unterbrechungen** oder **Interrupts**.

Erweiterungen der WüRC

Folgende Maßnahmen sind zur Erweiterung der WüRC nötig:

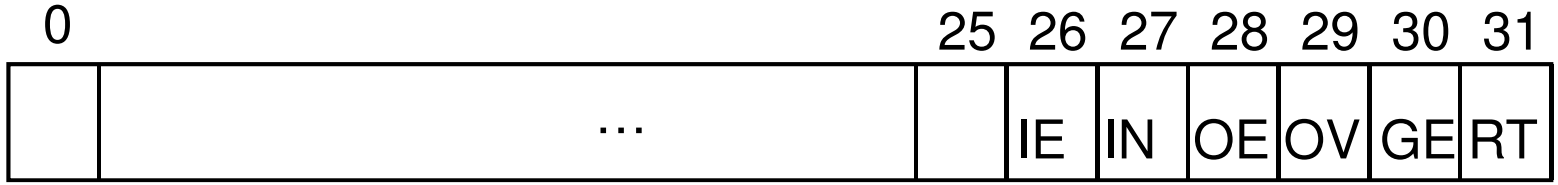
1. Wir geben dem Register R1 die Rolle eines Statusregisters. Es sollte nicht mehr zum Zwischenspeichern von Rechenergebnissen benutzt werden. Ferner werden Flags der Funktionseinheiten bei jeder Operation zum Statusregister „geodert“. Wir nennen R1 fortan **SR**.
2. Wir geben einem festen Register die Rolle des Stackpointers, zum Beispiel R2. Wir nennen es von nun an **SP**.
3. Wir führen von aussen eine Leitung **INT** in die CPU, die Ereignisse in der Peripherie mit *1* signalisiert.

Damit bleiben der WüRC nur noch ≤13 General Purpose Register.

Wir stellen uns vor, dass die Hardware, die die Peripherie steuert, seitens der CPU durch den Inhalt von Spezialregistern sichtbar und bedienbar ist. Diese Spezialregister liegen auf festen Adressen im Speicher (Memory Mapped I/O).

Das Statusregister

Wir stellen uns für das Statusregister zunächst mal folgende, sehr einfache Belegung vor. Da noch viel Platz im SR ist, sind auch viele Erweiterungen denkbar:



RT: Resetbit. SR[31] wird gesetzt, wenn eine illegale Instruktion dekodiert wurde. Kann nicht disabled werden.

GE: Global Interrupt Enable. Wird während der Behandlung von Unterbrechungen zurückgesetzt.

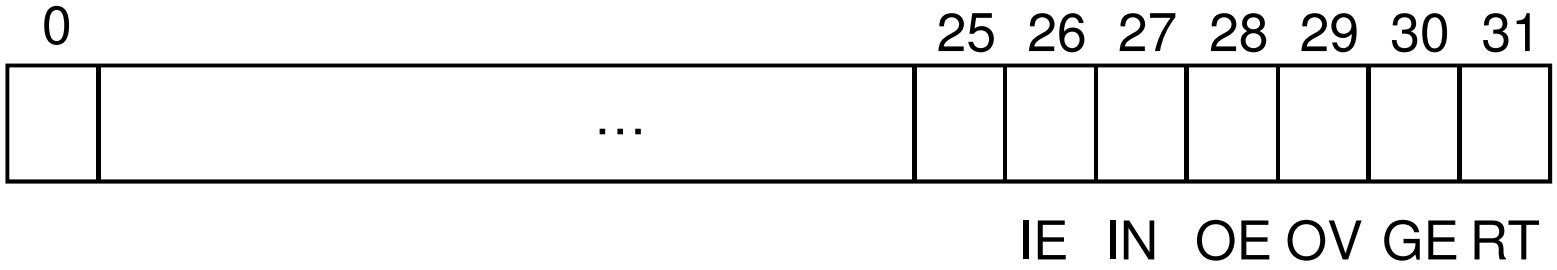
OV: Das Overflowbit wird stets in SR[29] nach jeder Operation „geodert“.

OE: Overflow Enable. Ist SR[28] = 1 wird durch ein gesetztes Overflowbit eine Exception ausgelöst, sofern SR[30] = 1 ist.

IN: Das externe INT Signal wird nach jedem Takt auf SR[27] dazu „geodert“.

IE: Interrupt Enable. Ist SR[26] = 1 wird durch ein gesetztes IN Bit eine Exception ausgelöst, sofern SR[30] = 1 ist.

Das Statusregister



Es macht Sinn, gleich zu allen diesen Flags folgende z.T. gleichnamige Konstanten zu definieren:

```
RT := 0x00000001;
GE := 0x00000002;
OV := 0x00000004;
OE := 0x00000008;
IN := 0x00000010;
IE := 0x00000020;
```

Auslösen von Unterbrechungen

Wir brauchen nun noch Befehle, die bei Annahme einer Unterbrechung den PC auf den Stack schieben, ohne ein Register zu verändern, und bei der Rückkehr den PC vom Stack holen, ohne ein Register zu verändern.

Wir nennen diese **`TRAP`** und **`RFE`**.

`TRAP` wird durch einen freien Code kodiert. Er bewirkt

- ein Store des PC auf den Stack und
- ein Dekrement des Stackpointers (Stack stets von MaxAdr abwärts!)
- ein Rücksetzen des GE-Bit im SR.
- ein Durchstarten auf Adresse 0.

`RFE` wird durch einen freien Code kodiert. Er bewirkt

- ein Laden des PC von Stackpointer + 1
- ein Inkrement des Stackpointers (Stack stets von MaxAdr abwärts!)
- ein Setzen des GE Bit im SR.

Auslösen von Unterbrechungen

Die Instruction Fetch Stufe schaut stets beim Laden des IR auch auf die Flags des Statusregisters.

Gilt

$$(GE \text{ and } ((OV \text{ and } OE) \text{ or } (IN \text{ and } IE) \text{ or } RT)) = 1$$

dann lädt die Instruction Fetch Stufe den Befehl **`TRAP`** ins IR. Dieser steht nicht im Programmspeicher, sondern wird von der Kontrolle der Instruction Fetch Stufe erzeugt.

Behandeln von Unterbrechungen

Ab Adresse 0 steht nun immer ein Programmstück, das zunächst den Fall eines echten Reset (SR = 0) oder einer angenommen Unterbrechung unterscheidet. Bei angenommenen Unterbrechungen werden diese in Software behandelt.

Wir skizzieren möglichen Code beispielhaft:

```
0:BNEZ SR,#1; -- Überspringe Programmstart bei INT
1:JMP #Programmstart;
2:SUBI SP,SP,#1;
3:STORE R3,SP,#1; -- Rette R3
4:ANDI R3,SR,#RT;
5:BEQZ R3,#ExeptOrInter;
Reset:AND SR,SR,R0; -- SR rücksetzen
7:…… -- Reset ggf behandeln* und dann neu starten
```

Behandeln von Unterbrechungen

```
ExeptOrInter:ANDI R3,SR,#OE;-- teste Overflow enable
BEQZ R3,#Interrupt -- Overflow disabled
ANDI R3,SR,#OV;
BEQZ R3,#Interrupt; -- 0 = keine Overflow Exception!
XORI SR,SR,#OV; -- Overflowflag rücksetzen
JAL R3,#Overflowhandler; -- Overflow behandeln*
JMP #Return; -- Rückkehrsequenz ausführen
-- Interrupt muss enabled sein und vorliegen
-- Interrupt behandeln:
Interrupt: JAL R3,#Interrupthandler; -- s.u.*
Return: LOAD R3,SP,#1; -- R3 Wiederherstellen
ADDI SP,SP,#1;
RFE; -- Rückkehr aus Exception
-- *) Achtung: alle Handler müssen in Callee Saves
-- mit R3 als Rücksprungübergabe implementiert sein!
```

Unterbrechungen Ausblick

Wir haben hier die einfachste Erweiterung betrachtet, damit wir nicht so viel ändern müssen.

In der Praxis können solche Konzepte komplexer sein. Ein paar Eckpunkte sind:

- Geschachtelte Unterbrechungen:

Wir haben während des Behandelns von Unterbrechungen keine weiteren Unterbrechungen mehr zugelassen. Das kann vor allem dann, wenn viel Peripherie über Interrupts zu bedienen ist, zu zu langen Reaktionszeiten führen. Also lässt man Unterbrechungen höherer Ebene zu, während man niedrigere Ebenen behandelt. (SR muss gesichert werden, TRAP #level statt TRAP, Level Enable Bits statt GE Bit….)

Unterbrechungen Ausblick

- Unterschiedliche Einsprungadressen:

Wenn man viele Ursachen für Interrupt oder Exception hat, müsste man in unserem Fall viel Zeit damit verbringen, in Falldiskussionen die Ursache herauszufinden und den richtigen Handler aufzurufen. Dies kann man vereinfachen, wenn man mehrere Einsprungadressen für TRAP Befehle hat, z.B. für jedes Level eine feste Adresse im Speicher.

In manchen Maschinen stehen diese Adressen in (einem oder mehreren) Spezialregistern, in anderen sind sie dem Level fest zugeordnet.

Unterbrechungen Ausblick

Häufig kann man die Empfindlichkeit gegenüber Ausnahme-situationen und Unterbrechungen vom Programm (oder System) aus einstellen, indem man Maskierungsbits setzt. Beispiele für weitere Unterbrechungsmechanismen sind

- Breakpoints
- Verschiedene Genauigkeitsgrade (z.B. Gleitkomma)
- Externe Signale
- Verschiedene Privilegstufen
- Systemaufrufe
- Speicherverwaltung,…