FACHHOCHSCHULE
TECHNIKUM WIEN

Fachbereich: Embedded Systems

Themengebiet: Embedded Systems

Interruptverarbeitung

Version 2.0, September 2005

Peter Balog

Inhaltsverzeichnis

0. Übersicht

0.1. Lehrziele, Lehrinhalte

Ziel des vorliegenden Studienbriefes "Interruptverarbeitung" ist die Vermittlung von allgemeinen, systemunabhängigen Kenntnissen über die Verarbeitung von *Hardware-Interrupts*. Bevor das Operationsprinzip einer CPU um die Fähigkeit der *Hardware-Interrupt*-Verarbeitung erweitert wird, wird im Kapitel 1 ein CPU-Modell abstrakt definiert.

0.2. Voraussetzungen

Prinzipielle Funktion und Programmiermodell eines Mikroprozessors.

0.3. Lernwegempfehlung

Versuchen Sie zuerst die lehrzielorientierten Fragen 1 bis 12 (Kapitel 4) zu beantworten. Danach arbeiten Sie Kapitel 1 und Kapitel 2 durch und versuchen die lehrzielorientierten Fragen 1 bis 12 und 13 bis 21 zu beantworten. Sollten Sie mit dem Kapitel 1 Schwierigkeiten haben, dann arbeiten Sie die relevanten Studienbriefe der Lehrveranstaltung „Digitale Systeme“ nochmals durch.

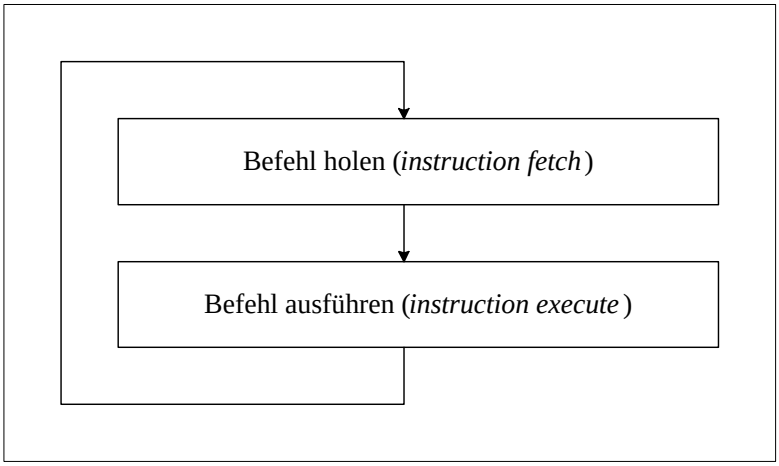
Das zentrale Kapitel dieses Studienbriefes ist das Kapitel 3. Nach dem Durcharbeiten sollten sie die lehrzielorientierten Fragen 22 bis 34 beantworten können.

0.4. Literatur

/1/ Flick, Liebig, „*Mikroprozessortechnik*“, 4. Auflage, 1994, Springer Lehrbuch, ISBN 3-540-57010-1

1. Einfaches CPU-Modell

Abstrahiert man die Arbeitsweise eines beliebigen Mikrocomputers, so erkennt man das zweiphasige Operationsprinzip, das auf John von Neumann zurückgeht (ca. 1940).



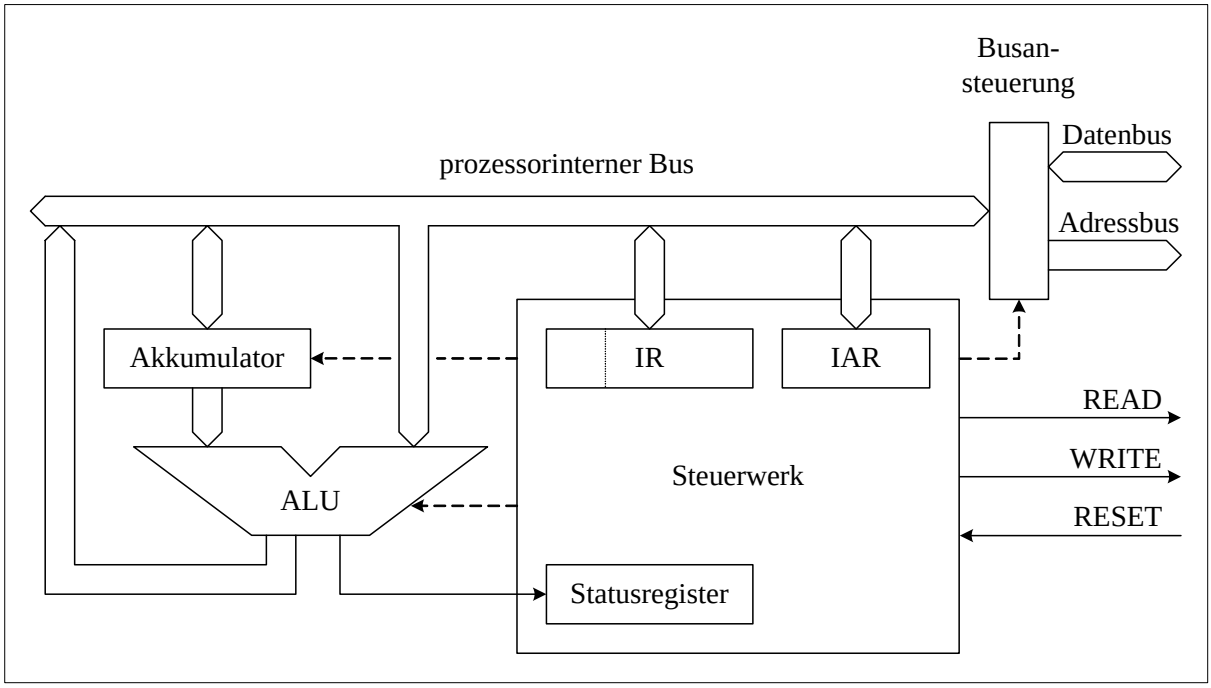
Die CPU muss zuerst einen Befehl holen (aus dem Speicher lesen) und kann ihn anschließend ausführen. Danach wird der nächste Befehl geholt. Programme werden somit streng sequentiell abgearbeitet. Ohne zusätzliche Einrichtungen gibt es keine Möglichkeit asynchron auf externe Ereignisse zu reagieren und somit gewissermaßen eine Parallelverarbeitung zu implementieren.

Wenn mit Hilfe einer “klassischen” Hochsprache (Pascal, C, C++, ...) ein Algorithmus in ein Programm umgesetzt wird, dann handelt es sich um die Automatisierung eines sequentiellen Vorgangs. Ein (sequentielles) Programm besteht aus einer linearen Abfolge von Anweisungen, aus Kontrollkonstrukten (bedingte und unbedingte Verzweigungen) und Funktionsaufrufen (Aufruf, Rücksprung). Das elementare CPU-Operationsprinzip (Zweiphasen-Verarbeitung) ist geeignet, solche Programme ablaufen zu lassen. Um ein (sequentielles) Hochsprachen-Programm auf eine CPU abzubilden, muss die CPU gewisse Betriebsmittel zur Verfügung stellen, die der Flusskontrolle des Programms dienen.

1.1. Notwendige CPU-Ressourcen

Die folgende Abbildung zeigt stark vereinfacht das Blockdiagramm einer 1-Adress-CPU. Mit Hilfe des Adressbus spricht die CPU Speicher oder Peripherie an. Über den bidirektionalen Datenbus können Daten gelesen (Richtung zur CPU) oder geschrieben (Richtung von der

CPU) werden. Die beiden Steuerleitungen `READ` und `WRITE` zeigen den Lese- bzw. den Schreibbetrieb an.



Der Akkumulator ist das zentrale Rechen- bzw. Arbeitsregister der 1-Adress-CPU in der Datenverarbeitungseinheit (ALU = *Arithmetic Logic Unit*). In der Befehlsverarbeitungseinheit (kurz Steuerwerk) sind folgende Ressourcen vorhanden:

- *Instruction Address Register* (IAR)
- *Instruction Register* (IR)

  speichert den zuletzt geladen Befehl während der Ausführung
- *Program Status Word* (PSW, Statusregister)
- *Stack*-Verwaltung (im Blockdiagramm nicht eingezeichnet)

1.1.1. Das *Instruction Address Register*

Das *Instruction-Address-Register* wird auch als *Instruction-Pointer* oder als *Program-Counter* bezeichnet. Nach jedem abgeschlossen Befehlszyklus enthält das IAR die Adresse des nächsten Befehls. Das IAR zeigt somit immer auf den nächsten auszuführenden Befehl.

Nach jeder *Fetch*-Phase wird das IAR automatisch erhöht, um so auf den nächsten Befehl in einer linearen Befehlssequenz zu zeigen. In der *Execute*-Phase von Verzweigungsbefehlen wird das IAR durch die Befehlsausführung selbst beeinflusst. Das Rücksetzen (*Reset*) der

CPU bewirkt, dass ein vordefinierter Anfangswert in das IAR geladen. An dieser sogenannten Reset-Adresse beginnt die Programmausführung nach einem Rücksetzvorgang.

1.1.2. Das *Program Status Word*

Das Ergebnis einer Operation der ALU (*Arithmetic Logic Unit*) wird durch sogenannte *Flags* qualifiziert. Sie dienen der Fehlererkennung, zur Realisierung von mehrstelligen Operationen und sind die Basis für bedingte Verzweigungen. Diese ALU-Flags werden auch als Bedingungsflags (*condition-flags, condition-codes*) bezeichnet,- die wichtigsten sind:

- Carry: Übertrag, vorzeichenloser Fehler
- Overflow: vorzeichenbehafteter Fehler
- Zero: Ergebnis enthält lauter 0-Bits
- Sign: Kopie des höchstwertigen Ergebnis-Bits

1.1.3. Der *Stack*

Der *Stack* (Stapelspeicher, Kellerspeicher) realisiert einen als LIFO (*Last In First Out*) organisierten Speicherbereich, welcher sich i.A. außerhalb der CPU im normalen Hauptspeicher befindet. Der *Stack* dient in erster Linie um verschachtelte Funktionsaufrufe realisieren zu können. Wird eine Funktion aufgerufen, so wird die Rücksprungadresse, das ist die Adresse des auf den Funktionsaufruf folgenden Befehls, auf dem *Stack* abgelegt. Die Funktion muss mit einem speziellen Rücksprungbefehl beendet werden, der den obersten Stackeintrag (TOS ... *Top Of Stack*) entfernt und als Sprungziel interpretiert.

Der *Stack* wird über das spezielle CPU-Register SP (*Stack Pointer*) und die beiden Methoden `PUSH` und `POP` verwaltet. Diese beiden Methoden werden einerseits implizit in den Befehlen zum Funktionsaufruf (`call`) und Rücksprung (`return`) verwendet und andererseits sind sie meist als explizite `push`- und `pop`-Befehle implementiert, um Daten mit dem *Stack* auszutauschen. Bevor ein Programm den *Stack* sinnvoll nutzen kann, muss ein Speicherbereich für den *Stack* reserviert und der *Stack-Pointer* entsprechend initialisiert werden.

Anmerkung:

> In heute verwendeten Hochsprachen wird der *Stack* zusätzlich für die lokalen Variablen und die Parameterübergabe an Funktionen verwendet. Damit in einer Funktion auf die lokalen Variablen und die Übergabeparameter zugegriffen werden kann benötigt man als weitere CPU-Ressource den *Frame-Pointer*, um diesen Stackbereich (*Stack-Frame*) zu adressieren. Der *Frame-Pointer* alleine genügt natürlich nicht,- die CPU muss auch die “indirekte Adressieung mit Offset” unterstützen.

1.2. Das elementare CPU Operationsprinzip

Das folgende Programm in einer Pseudosyntax beschreibt die Funktion einer CPU auf höchster Abstraktionsebene:

```
reset:      IAR=<reset_address>;
            loop
                 IR=mem(IAR);
                 IAR=IAR+1;
                 exec(IR);
            end loop;
```

Der *Reset* kann völlig asynchron zur Befehlsausführung erfolgen. Der *Reset* stellt eine Unterbrechung der höchsten Priorität dar, von der es keine Rückkehr zur Unterbrechungsstelle gibt. Die Programmausführung beginnt somit wiederum bei „Null“.

Der *Reset* initialisiert zumindest das IAR,- danach beginnt an dieser Stelle die Programmabarbeitung gemäß dem Operationsprinzip (*fetch, execute*). Der Speicherinhalt jener Speicherstelle die durch das IAR adressiert wird, wird in das IR der CPU geladen. Da das IAR durch die Befehlsausführung verändert werden kann (Verzweigung), muss die Erhöhung um 1, was nichts anderes bedeutet als auf den nächsten Befehl in einer linearen Befehlssequenz zu zeigen, vor der Ausführung (`exec`) erfolgen.

1.2.1. Die Operation `mem()`

Die `mem`-Operation implementiert den elementaren Speicherzugriff. Abhängig davon ob `mem()` rechts oder links vom „=“ steht wird eine Lese- bzw. eine Schreibe-Operation ausgeführt.

```
item = mem(address):
        {CPU_address_bus} = address;
        CPU_read = <active>;
        wait some time;
        item = {CPU_data_bus};
        CPU_read = <inactive>;
```

Bei einem lesenden Zugriff wird zuerst die Adresse auf den Adressbus der CPU ausgegeben. Danach aktiviert der Prozessor das Steuersignal `read` und wartet anschließend eine gewisse vordefinierte Zeit. Diese Zeit benötigt der Speicher um die angeforderten Daten bereitzustellen. Die CPU liest diese Daten über den Datenbus ein und deaktiviert das `read` Steuersignal.

```
mem(address) = item:
        {CPU_address_bus} = address;
        {CPU_data_bus} = item;
        CPU_write = <active>;
        wait some time;
        CPU_write = <inactive>;
        {CPU_data_bus}=<inactive>;
```

Bei einem Schreibzugriff müssen sowohl der Adressbus als auch der Datenbus von der CPU angesteuert werden. Mit dem Steuersignal `write` wird dem Speicher signalisiert, dass er Daten übernehmen kann.

1.2.2. Die Operation `exec()`

Diese Operation führt einen CPU-Befehl aus. Das Argument der Funktion `exec()` ist der binär codierte Maschinenbefehl wie er z.B. im IR nach einem *Instruction Fetch* vorliegt.

1.3. CPU-Befehle

CPU-Befehle setzen sich aus elementaren Teilfunktionen zusammen. Die Funktion von CPU-Befehlen lässt sich ebenfalls in einer Pseudosyntax beschreiben. Einige, für das weitere Verständnis notwendigen, Befehle werden kurz vorgestellt.

1.3.1. PUSH- und POP

Die PUSH- und POP-Methoden, bzw. die Befehle `push` und `pop` dienen dem Datentransfer mit dem *Stack*. Üblicherweise wächst der *Stack* von hohen zu niedrigen Adressen und der SP zeigt stets auf den letzten *Stack*-Eintrag (*top of stack*). Damit ergeben sich die folgenden Funktionsbeschreibugen:

```
push(item):
        SP = SP-1;
        mem(SP) = item;
```

```
pop(item):
        item = mem(SP);
        SP = SP+1;
```

Referenziert `item` ein Objekt das breiter als die Verarbeitungsbreite der CPU ist, werden meherere Speicherzyklen benötigt. Die meisten CPU-Implementierungen unterstützen diese Fähigkeit. Z.B. erlaubt eine 8-Bit CPU das Verarbeiten von 8-, 16- und 32-Bit Objekten, was zu 1-, 2- und 4-fachen Speicherzyklen führt. Diese mehrfachen Speicherzyklen sind für den Befehl “transparent”, d.h. sie werden automatisch in der *Execute*-Phase realisiert.

1.3.2. Bedingte Verzweigung

Bei Verzweigungen wird der Wert des IAR durch die Befehlsexekution gezielt verändert. Bei bedingten Verzweigungen erfolgt dies nur, wenn die Bedingung erfüllt ist.

```
jump cond,addr:
        if <cond is true> then
             IAR = addr;
        end if ;
```

Die Bedingung (`cond`) kann je nach CPU-Implementierung lediglich direkt ein *Condition-Flag* (z.B.: Carry) des PSWs sein, oder aber eine komplexe Bedingung, wo CPU-intern mehrere Condition-Flags geeignet kombiniert werden (z.B.: *signed greater equal*). Eine unbedingte Verzweigung ist normalerweise direkt im CPU-Befehlssatz enthalten (`jump addr`),- wenn nicht, dann muss eine "*always true condition*" eingesetzt werden.

1.3.3. Funktionen (`call` und `return`)

Funktionen einer Hochsprache werden über den Unterprogramm-Mechanismus auf die Maschinenebene abgebildet. Bevor in die Funktion verzweigt wird, wird die Rücksprung-adresse auf dem *Stack* abgelegt. Die `PUSH`- und die `POP`-Methoden werden implizit verwendet.

```
call addr:                    return:
        push(IAR);                    pop(IAR);
        IAR = addr;
```

Der `call`-Befehl realisiert eine Verzweigung mit Gedächtnis,- d.h. dass die Adresse des auf den `call`-Befehl folgenden Befehls am *Stack* abgelegt wird. Der `return`-Befehl ist ein sogenannter impliziter Verzweigungsbefehl,- das bedeutet, dass kein Sprungziel spezifiziert wird. Der `return`-Befehl entfernt den obersten *Stack*-Eintrag und lädt damit das IAR.

2. Interruptverarbeitung

Die Verarbeitung von asynchronen Ereignissen sowie die Idee der Parallelverarbeitung motivieren die Erweiterung des Operationsprinzips einer CPU um die Fähigkeit der Interruptverarbeitung.

2.1. Asynchrones Ereignis

Asynchrone Ereignisse können mit einer CPU ohne (Hardware-) Interruptfähigkeit nur mit Hilfe von *Polling* verarbeitet werden. *Polling* bedeutet das dauernde bzw. periodische Abfragen der Ereignisquelle gefolgt von einer bedingten Verzweigung.

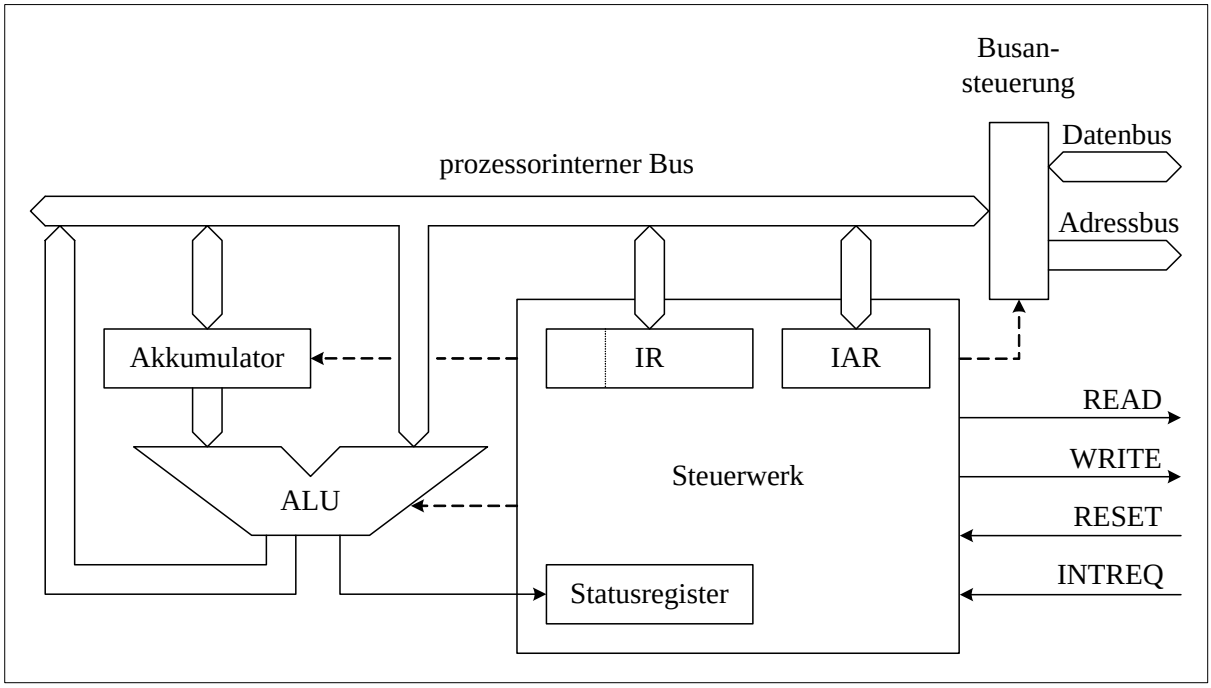
```
while (EVENT_SOURCE != ACTIVE)
   { <do_nothing> } ;
<do_event_processing>;
...
```

In obigem Beispiel bleibt das Programm solange in der `while`-Schleife hängen, solange die Ereignisquelle inaktiv ist. Erst wenn das Ereignis eingetreten ist, dann wird die Schleife verlassen und das Programm führt die Ereignisverarbeitung durch. In der Warteschleife kann normalerweise nichts Sinnvolles durchgeführt werden. Es wird lediglich CPU-Zeit vernichtet. Das wiederholte Abfragen einer (asynchronen) Ereignisquelle ist somit genauso uneffektiv wie das dauernde Abheben eines Telefons um festzustellen ob man gerade angerufen wird.

Man muss dem asynchronen Ereignis ein “Signal” zuordnen, mit dem das laufende Programm unterbrochen und zu einer „Ereignis-Verarbeitungs-Routine“ verzweigt wird. Nach der „Ereignis-Verarbeitung“ wird mit dem unterbrochenen Programm weitergemacht. Das Operationsprinzip der CPU muss also derart erweitert werden, dass mit einem externen Signal ein Unterprogramm gestartet werden kann.

2.2. Einfacher Hardware-Interrupt

Vorerst betrachten wir eine einfache CPU, die nur einen einzigen Hardware-Interrupt verarbeiten kann, der über die Steuerleitung INTREQ signalisiert wird. Wir werden dazu das CPU-Operationsprinzip, aber auch die CPU-Ressourcen erweitern müssen.



Wenn der HW-Interrupt eintrifft (INTREQ wird aktiv), dann verzweigt die Programmverarbeitung zu einer Funktion, welche an einer vordefinierten Stelle im Hauptspeicher liegt. Diese spezielle Funktion wird als *Interrupt Service Routine* (ISR) bezeichnet. Der HW-Interrupt realisiert einen von extern ausgelösten, somit asynchronen Funktionsaufruf. Das erweiterte Operationsprinzip lautet:

```
reset:      IAR = <reset_address>;
            loop
                 IR = mem(IAR);
                 IAR = IAR+1;
                 exec(IR);
                 if INTREQ then
                      push(IAR);
                      IAR = ISR_addr;
                 end if;
            end loop;
```

Aus dem Operationsprinzip erkennt man, dass ein Interrupt nie die Befehlsausführung unterbrechen kann. Eine Unterbrechung kann also nur “zwischen” Befehlen erfolgen,- oder anders formuliert:

> Der Befehlszyklus ist bezüglich des Hardware-Interrupts unteilbar!

Die Operationssequenz innerhalb der `if`-Anweisung wird als *Interrupt-Processing* bezeichnet.

2.3. Probleme mit dem erweiterten Operationsprinzip

Das vorgestellte Operationsprinzip lässt zwei Probleme erkennen. Einerseits kann es sinnvoll sein, dass während zeitkritischen Programmsequenzen die CPU nicht auf INTREQ reagieren soll. Andererseits kann ein zu lang andauerndes INTREQ-Signal zu einem rekursiven ISR-Aufruf führen, da ja bereits im ersten Befehl der ISR das INTREQ-Signal wieder abgefragt wird.

Anmerkung:

> Bei einem flankensensitiven INTREQ-Eingang kann das Problem von rekursiven Aufrufen nicht auftreten.

In beiden Fällen liegt die Lösung in der Implementierung einer INTREQ-Sperrfunktion.

2.4. Das Interrupt-Flag

Diese Sperrfunktion wird mittels des *Interrupt-Flags* (IF) realisiert. Das IF ist ein Steuerbit (*control flag*) im PSW. Ist IF=’1’ dann ist die CPU auf INTREQ sensibel (*interrupt enabled*). Bei IF=’0’ ist das INTREQ Signal gesperrt (*interrupt disabled*) Das IF kann entweder explizit über zusätzliche Befehle (`enable_int`, `disable_int`) beeinflusst werden, oder es wird implizit während des *Interrupt-Processings* rückgesetzt. Das Operationsprinzip muss wie folgt erweitert werden:

```
reset:      IAR = <reset_address>;
            loop
                 IR = mem(IAR);
                 IAR = IAR+1;
                 exec(IR);
                 if (INTREQ and PSW.IF='1') then
                      push(IAR);
                      push(PSW);
                      PSW.IF = '0';
                      IAR = ISR_addr;
                 end if;
            end loop;
```

Da während des *Interrupt-Processings* der Inhalt des PSW verändert wird, wird auch das PSW wie das IAR auf dem *Stack* gesichert. Die CPU sichert also den sogenannten elementaren CPU-Kontext (IAR, PSW) automatisch.

Das so erweiterte Operationsprinzip macht es wiederum notwendig, dass ein spezieller Rücksprungbefehl für ISRs implementiert werden muss.

2.5. Der Befehl *Return from Interrupt*

Die ISR muss mit dem speziellen *Return*-Befehl `iret` abgeschlossen werden. Der Befehl `iret` wird CPU-intern wie folgt implementiert :

```
iret:
        pop(PSW);
        pop(IAR);
```

Voreilig könnte man meinen, dass es kein Problem ist, wenn der Prozessor-Befehlssatz keinen `iret`-Befehl enthält. Dann muss die Funktion eben wie folgt ausprogrammiert werden:

```
pop(PSW)
return
```

Das Ausprogrammieren birgt jedoch eine Gefahr in sich:

Mit dem Befehl `pop(PSW)` wird der alte Wert des `PSW.IF='1'` wieder hergestellt,- d.h. INTREQ ist wieder *enabled*. Gemäß der Operationsvorschrift erfolgt nach dem

```
exec(pop(PSW))
```

die Abfrage

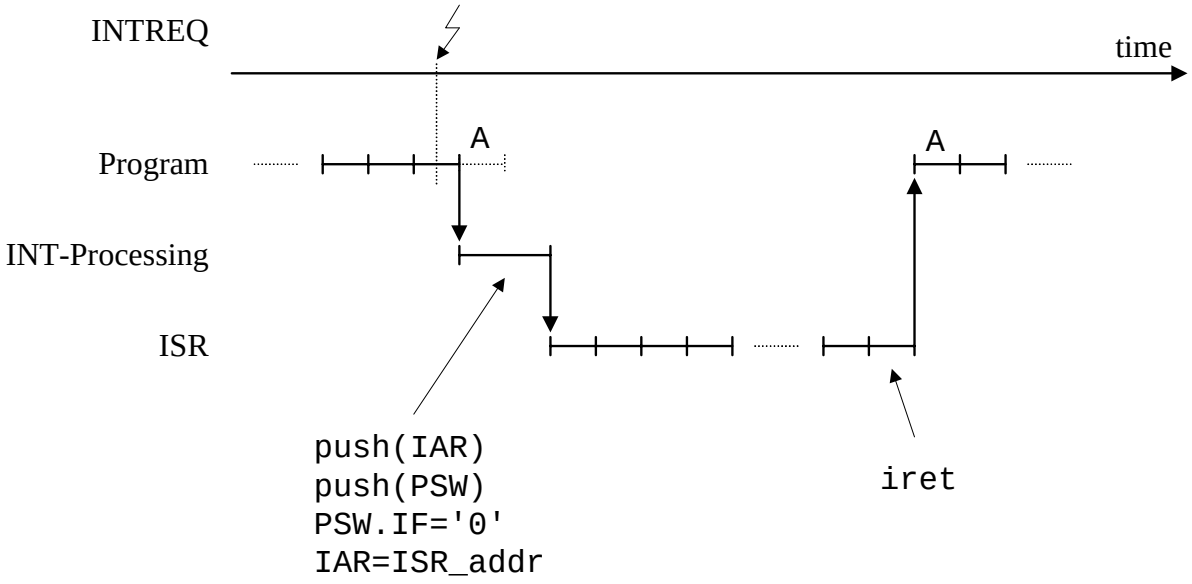
```
if (INTREQ and PSW.IF='1') then ...
```

Liegt nun bereits ein neuer INTREQ an, dann ist die Bedingung erfüllt, ein weiteres *Interrupt-Processing* wird eingeleitet und der Befehl `return` wird (zumindest derzeit) nicht ausgeführt. Logisch gesehen kann nichts passieren, da der elementare CPU-Kontext (IAR, PSW) automatisch auf dem *Stack* gesichert wird. Tritt dieses Szenario jedoch in einer ununterbrochen zeitlichen Reihenfolge von ausprogrammierten "*Return from Interrupts*" auf, kann ein *Stack-Overflow* das System zum Absturz bringen.

2.6. Zeitlicher Ablauf

Die Zeit vom Eintreffen des Interrupts bis zur 1. Instruktion der *Interrupt-Service-Routine* (ISR) wird als Interrupt-Latenz-Zeit bezeichnet. Ihr ungünstigster Wert ergibt sich für ein einfaches System zu:

Latenz-Zeit = Zeit des längsten Befehls + Zeit für *Interrupt-Processing*



Die Abbildung zeigt sehr schön, dass die ISR parallel zum unterbrochenen Programm ausgeführt wird. Und genau dort liegt ein großes Problempotential, wenn die ISR nicht "transparent" ist.

Aufgabe 1:

Zeichnen Sie in der vorigen Abbildung die Latenz-Zeit ein.

Anmerkung:

Natürlich läuft die ISR nicht gleichzeitig zum Programm ab,- vielmehr sind die ISR und das Programm zeitlich verschachtelt. Dies ändert jedoch nichts daran, dass die ISR und das Programm als parallel zueinander aufgefasst werden müssen. Ist die ISR nicht "transparent" und verändert z.B. ein CPU-Register, dann ***kann*** das unterbrochene Programm beeinflusst werden,- abhängig davon ob gerade das Register vom Programm verwendet wird oder nicht. Da das Eintreten des Interrupts und damit die Ausführung der ISR vollkommen asynchron zur "normalen" Programmausführung ist, kann das Ergebnis des Programms bei identen Eingangsdaten von Lauf zu Lauf unterschiedlich sein (Indeterminismus).

2.7. Aufbau der ISR

Eine ISR ist dann transparent, wenn das unterbrochene Programm von der Unterbrechung nichts merkt. Der elementare CPU-Kontext (IAR, PSW) wird implizit durch das Operationsprinzip gesichert. Der weitere CPU-Kontext (Register, etc.) muss explizit in der ISR gesichert werden. Aus diesen Überlegungen folgt der allgemeine Aufbau einer ISR:

- Sichern der CPU-Register, etc. (push)
- "Ruhigstellen" der Interrupt-Quelle

  damit das INTREQ-Signal wieder inaktiv wird
- die eigentliche Funktion der ISR
- Restaurieren der CPU-Register, etc. (pop)
- Abschluss mit iret

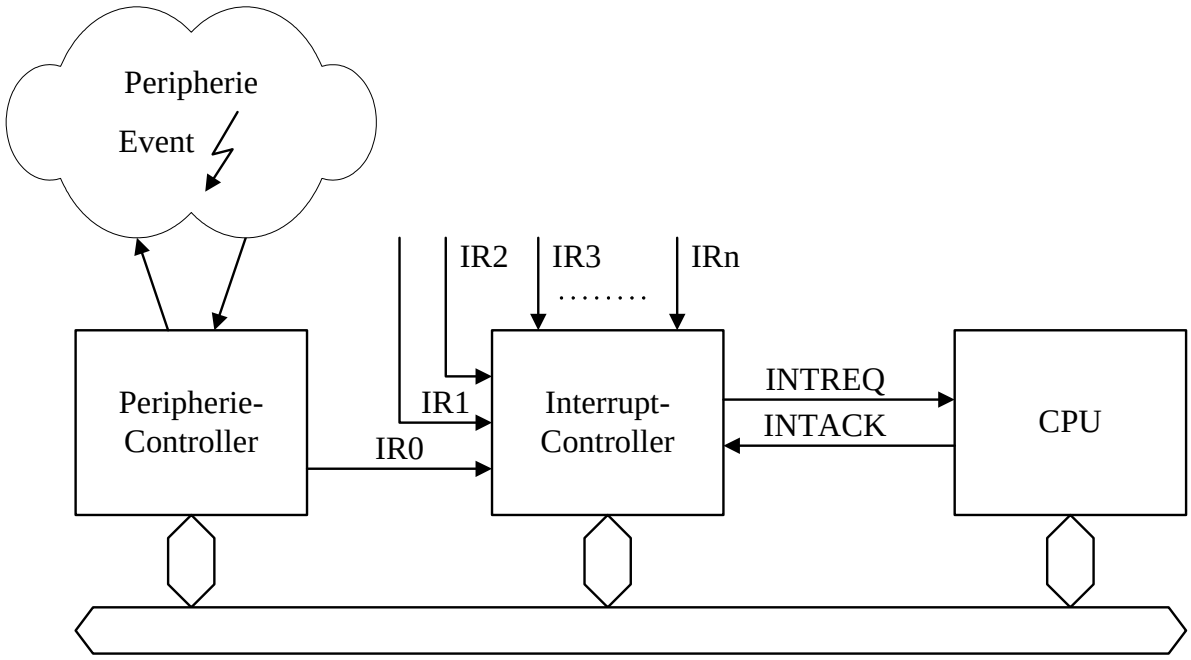
3. Vektorielles Interruptkonzept

Soll eine CPU mehrere Interruptquellen verarbeiten können gibt es die einfache, jedoch sehr unflexible Möglichkeit, die CPU mit mehreren INTREQ-Eingängen zu versehen. Für jeden $INTREQ_i$-Eingang ist eine Startadresse für die ISR_i a priori definiert. Dieses Konzept findet man noch heute bei eher „kleinen“ Mikrocontrollern.

Wir wollen uns in diesem Kapitel mit einem weitaus flexibleren Interruptkonzept befassen. Die CPU selbst hat nach wie vor nur einen INTREQ-Eingang. Eine akzeptierte Interruptanforderung löst jedoch in der CPU einen *Interrupt-Acknowledge*-Zyklus aus, in dem sich die Interruptquelle identifizieren muss. Abhängig von der Identifikation verzweigt die CPU zur richtigen ISR. Die Verzweigung erfolgt somit indirekt.

In heute verwendeten Systemen zentralisiert ein *Interrupt-Controller* alle Interruptanforderungen und wickelt die Kommunikation (INTREQ / INTACK Handshake) mit der CPU ab.

3.1. Der Interrupt-Signalpfad



Der Peripherie-*Controller* erzeugt als Folge eines Peripherie-*Events* einen *Interrupt-Request* (IR), der an den *Interrupt-Controller* weitergeleitet wird. Der *Interrupt-Controller* erzeugt einen INTREQ und leitet ihn an die CPU weiter. Die CPU quittiert den INTREQ mit einem

INTACK (diesen Vorgang nennt man *Handshake*),- der *Interrupt-Controller* schickt eine Identifikation, die **Interrupt-Vektor-Nummer,** (IVN) über den Datenbus an die CPU. Die CPU liest die Vektornummer und verwendet diese als Index in der **Interrupt-Vektor-Tabelle** (IVT). Das so adressierte Tabellenelement enthält die Startadresse der *Interrupt-Service-Routine*, zu der jetzt verzweigt wird („indizierte ISR-Adressierung“).

Die ISR sichert den CPU-Kontext, “beruhigt” den Peripherie-*Controller*, führt dann die eigentliche Funktion aus, restauriert den CPU-Kontext, sendet ein “End-Of-Interrupt” (EOI) an den *Interrupt-Controller* und endet mit der `iret`-Instruktion.

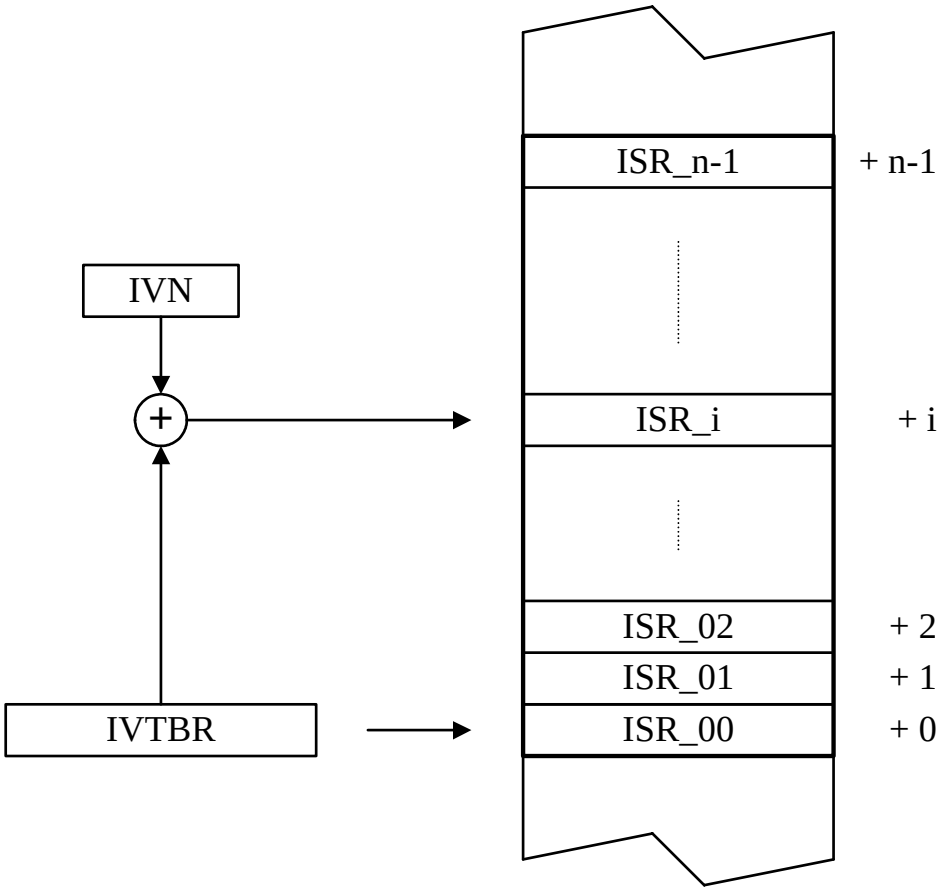
Damit der beschriebene Vorgang funktioniert, müssen zu Beginn der Applikation der Peripherie-Controller und der Interrupt-Controller korrekt programmiert werden. Weiters muss die Interrupt-Vektor-Tabelle initialisiert werden und die ISRs müssen an den richtigen Adressen liegen. Abschließend muss noch das PSW.IF=‘1’ gesetzt werden (mit dem `enable-int` Befehl).

Der Interrupt-Controller ist ein „intelligenter“ *Interrupt-Multiplexer*, bei dem das Verhalten der IR-Eingänge flexibel programmiert werden kann. So kann jeder IR-Eingang individuell maskiert (*enable* / *disable*) werden. Weiters kann zwischen Pegel- und Flankensensitivität gewählt, und die Polarität (*high* und *low*, bzw. *rising* und *falling*) definiert werden. Treten mehrere IRs gleichzeitig auf, dann priorisiert der Interrupt-Controller die Anfragen nach bestimmten programmierbaren Strategien (fixe Priorität, rotierende Priorität). Für jeden IR-Eingang verwaltet der Interrupt-Controller eine eindeutige Vektornummer.

3.2. Die Interrupt Vektor Tabelle

Wenn die CPU einen INTREQ akzeptiert, dann wird ein *Interrupt-Acknowledge*-Zyklus (INTACK aktiv) ausgeführt. Die CPU liest über den Datenbus die Interrupt-Vektor-Nummer (IVN) ein. Diese IVN wird als Index in der Interrupt-Vektor-Tabelle (IVT) verwendet.

Die IVT enthält die **Interrupt-Vektoren**,- das sind die Startadressen der Interrupt-Service-Routinen. Die Größe der IVT richtet sich nach den möglichen Vektornummern (typ. 256). Die IVT liegt i.A. im Speicher und wird über das IVT-*Base-Register* (IVTBR) adressiert (siehe folgende Abbildung).



Wenn die Startadressen der Interrupt-Service-Routinen breiter sind als die Speicherbreite (z.B. 64-Bit Adressen bei einem 32-Bit Prozessor), dann belegt jeder Vektor mehrere Speicherworte. Die Addition des Index zur Basis muss dann natürlich skaliert erfolgen, wobei die Skalierung das Verhältnis von Vektorbreite zu Speicherwortbreite darstellt.

Anmerkung:

> Dieses Prinzip einer skalierten Indizierung ist z.B. von der Programmiersprache C her bekannt. Die Addition eines Index zu einem Zeiger bzw. einer Adresse wird mit `sizeof(object)` skaliert. Nur so ist zu verstehen, dass ein Pointer nicht als Pointer definiert wird, sondern über den Objekt-Typ auf den er zeigt!

3.3. Erweitertes CPU Operationsprinzip

Die folgende Beschreibung des Operationsprinzips der CPU wurde um die Interrupt-Vektor-Verarbeitung erweitert:

```
reset:      IAR = <reset_address>;
            loop
                 IR = mem(IAR);
                 IAR = IAR+1;
                 exec(IR);
                 if (INTREQ and PSW.IF='1') then
                      IVN = getivn();
                      push(IAR);
                      push(PSW);
                      PSW.IF = '0';
                      IAR = mem(IVTBR+IVN);
                 end if;
            end loop;
```

Die neue eingeführte Operation `getivn()` ist für den *Interrupt-Achknowledge-Cycle* verantwortlich und liefert als Ergebnis die mit der Interruptquelle assoziierte Vektornummer. Für `getivn()` sind 2 Varianten der Implementierung möglich.

3.3.1. Variante 1 der Operation `getivn()`

Diese einfache Variante aktiviert das `INTACK`-Signal, wartet eine gewisse vordefinierte Zeit um anschließend über den Datenbus die IVN zu lesen. Nachdem Lesen wird `INTACK` wieder inaktiv.

```
getivn():
        INTACK = '1';
        wait some time;
        IVN = {CPU_data_bus};
        INTACK = '0';
```

Im Prinzip handelt es sich um einen speziellen Lesezyklus (vgl. dazu Kapitel 1.2.1).

3.3.2. Variante 2 der Operation `getivn()`

Die Variante 2 wartet nach dem Lesen der IVN auf die Zurücknahme der `INTREQ`-Signals, welches das aktuelle *Interrupt-Processing* ausgelöst hat.

```
getivn():
        INTACK = '1';
        wait some time;
        IVN = {CPU_data_bus};
        wait for INTREQ = '0';
        INTACK = '0';
```

Die Variante 2 implementiert ein sogenanntes 4-Phasen-Handshake. Dadurch wird sichergestellt, dass der aktuelle INTREQ inaktiv wird, bevor die Interruptverarbeitung mit der Verzweigung in die *Interrupt-Service-Routine* fortgesetzt wird.

Anmerkung:

> Oft ist die Variante 1 implementiert, da sie etwas schneller ist. Die folgende Randbedingung ist jedoch notwendig:
> Ist die Variante 1 der `getivn()`-Operation implementiert so muss sicher gestellt werden, dass der Interrupt-Controller die `INTREQ`-Leitung für den gerade bestätigten *Interrupt-Request* inaktiv gesetzt hat, bevor die CPU Interrupts wieder zulässt.

3.4. Funktion des Interrupt-Controllers

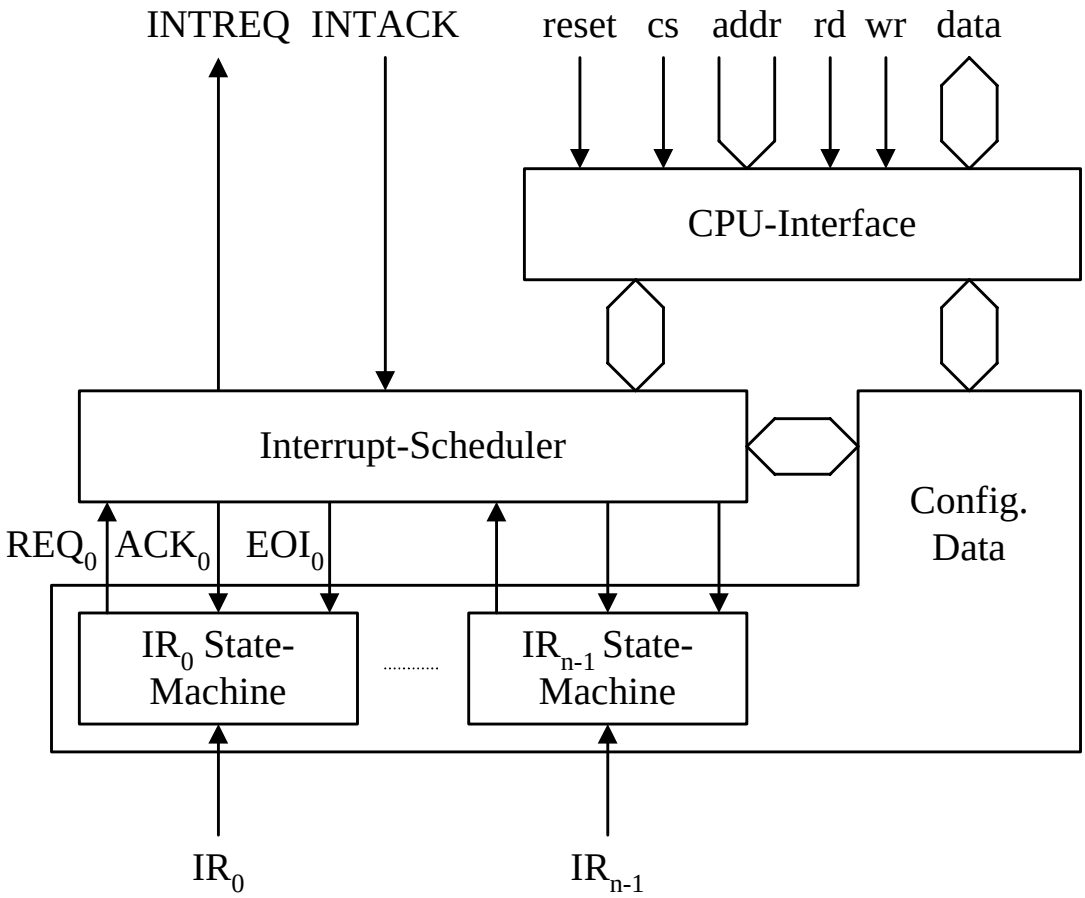
Der Interrupt-Controller ist ein programmierbarer Peripherie-Controller der mehrere *Interrupt-Request* (IR) Eingänge verwaltet und somit die CPU entlastet. Er verfügt über ein Standard CPU-Interface zur Programmierung (Konfiguration), Statusabfrage und für die Übertragung der Vektornummern (IVN). Für jeden IR Eingang ist eine *IR-State-Machine* implementiert. Das Herzstück ist der *Interrupt-Scheduler*, der im Falle von mehreren aktiven IRs den richtigen zur CPU durchschaltet.

Bevor der Interrupt-Controller seine Funktion wahrnehmen kann, muss er, wie jeder andere Peripherie-Controller auch, programmiert werden. Der Interrupt-Controller verwaltet für

jeden IR-Eingang einen strukturierten Konfigurationsdatensatz. Neben dieser IR-spezifischen Parametrierung ist noch eine globale Konfiguration notwendig, bei der z.B. die *Scheduling-Strategie* (statische Priorität, rotierende Priorität, etc.) definiert wird.

3.4.1. Blockdiagramm

Der Interrupt-Controller verfügt, wie jeder Peripherie-Controller, über ein Standard-CPU-Interface. Über dieses wird der Interrupt-Controller (i) programmiert, (ii) kann der Status abgefragt werden und (iii) werden die Vektor-Nummern während eines INTACK-Zyklus zur CPU übertragen.

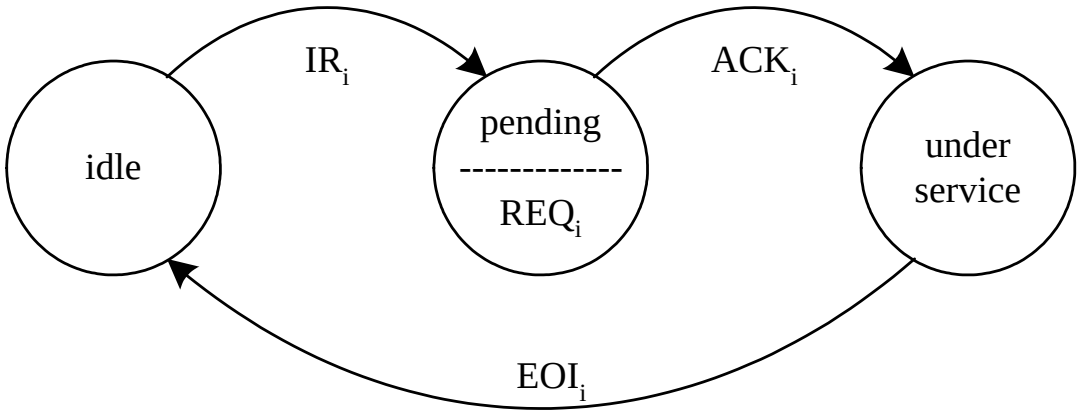


Bei der Programmierung werden die Konfigurationsdaten für jeden IR-Eingang initialisiert und die *Scheduling-Strategie* festgelegt.

3.4.2. Die IR-State-Machine

Wird der IR_i-Eingang aktiv, dann wechselt die IR_i-State-Machine vom Zustand `idle` in den Zustand `pending` und aktiviert die REQ_i-Leitung um dem Scheduler die Interruptanforderung zu melden.

Abhängig von der Gesamtsituation des Interruptsystems wird der Scheduler die Anforderung sofort oder erst später zur CPU weiterleiten.



Der Scheduler aktiviert infolge dieses REQ_i die `INTREQ`-Leitung zur CPU. Diese quittiert mit `INTACK`,- der Scheduler veranlasst das Senden der passenden Vektor-Nummer VN_i und signalisiert der IR_i-State-Machine ein ACK_i. Die IR_i-State-Machine deaktiviert den REQ_i und geht in den Zustand `under_service`,- sie "weiß" jetzt, dass die CPU ihre Anforderung bearbeitet. Solange eine Anforderung *under service* ist, kann über den IR_i Eingang keine neue Anforderung erzeugt werden. Daher muss in der Interrupt-Service-Routine vor dem `iret`-Befehl der Interrupt-Controller von der Beendigung des Services benachrichtigt werden. Das geschieht mit einer EOI-Benachrichtigung (*End-Of-Interrupt*), die an den Interrupt-Controller geschickt wird. Der Scheduler signalisiert dies der IR_i-State-Machine via EOI_i, die damit wieder in den Zustand `idle` wechselt und somit für neue Anforderungen bereit ist.

Anmerkung:

> Die EOI-Benachrichtigung kann bei sehr eng gekoppelten CPU / Interrupt-Controller Systemen (wie z.B. in Mikrocontrollern oder anderen *System-on-Chip* Lösungen) direkt in Hardware realisiert sein. Die dafür notwendige Funktionalität wird in den `iret` Befehl integriert.

3.4.3. Der Interrupt-Scheduler

Der Interrupt-Scheduler verwaltet die IR-State-Machines (`REQ`$_i$, `ACK`$_i$, `EOI`$_i$) und steuert die Interrupt bezogene Kommunikation mit der CPU (`INTREQ`, `INTACK`, `IVN`, `EOI-Nachrichten`). Bei mehreren aktiven Anforderungen priorisiert der Scheduler gemäß Programmierung. Üblich sind eine fixe Prioritätsvergabe oder eine rotierende Priorität. Über die Konfiguration können IR-Eingänge maskiert werden. Anforderungen auf maskierten Eingängen werden vom Scheduler ignoriert.

3.4.4. Der IR-Konfigurationsdatenvektor

Der Interrupt-Controller verwaltet einen Konfigurationsdatenvektor, der für jeden IR-Eingang ein strukturiertes Element enthält. Die IR-Nummer kann als Index in dem Vektor aufgefasst werden.

	Mask	Pending	Under Service	Priority	Vector Number	Properties
IR_0	enabled	0	0	5	12	rising
IR_1	disabled	0	0	3	13	low
IR_{N-1}	enabled	1	0	7	45	high

Die Strukturkomponenten haben die folgende Bedeutung:

Mask:	enable / disable IR
Pending:	IR-State-Machine ist im Zustand `pending`
Under Serv.:	IR-State-Machine ist im Zustand `under_service`
Priority:	Priorität des IRs
Vector Nr.:	Interrupt-Vektor-Nummer
Properties:	weitere Eigenschaften, wie Pegel, Flanke, Polarität, etc.

Diese Konfigurationsdaten können per Programm ausgelesen werden, einige können auch geschrieben werden. Die Informationen “Pending” und “Under Service” können i.a. nicht

geschrieben werden. Wird das System initialisiert, dann müssen für alle verwendeten IRs “Mask”, “Priority”, “Vector-Nr.” und “Properties” entsprechend gesetzt werden.

3.5. Interrupts ohne Verschachtelung

Wenn eine Interruptanforderung eine gerade laufende Interrupt-Service-Routine nicht unterbrechen darf, dann spricht man von einem System ohne „Verschachtelung“ (*without nesting*). Bei diesem einfachen, aber sehr effizientem Modell, besonders in Kombination mit einem *Multitasking-* oder *Real-Time-Operating-System*, sind Interrupt-Service-Routinen nie parallel zu einander. Bevor eine neue ISR starten kann, muss die gerade laufende ISR finalisiert werden. Die ISR ist somit selbst bzgl. des Interrupts unteilbar!

3.5.1. Aufbau der ISR (*without nesting*)

Während des Interrupt-Processings wird das PSW.IF auf 0 gesetzt. Damit werden alle weiteren Interruptanforderungen von der CPU ignoriert. In der ISR darf das PSW.IF nicht auf 1 gesetzt werden!

- Sichern der CPU-Register, etc. (push)
- “Ruhigstellen” der Interrupt-Quelle

  damit das IR-Signal wieder inaktiv wird
- die eigentliche Funktion der ISR
- Restaurieren der CPU-Register, etc. (pop)
- EOI-Nachricht an den Interrupt-Controller
- Abschluss mit `iret`

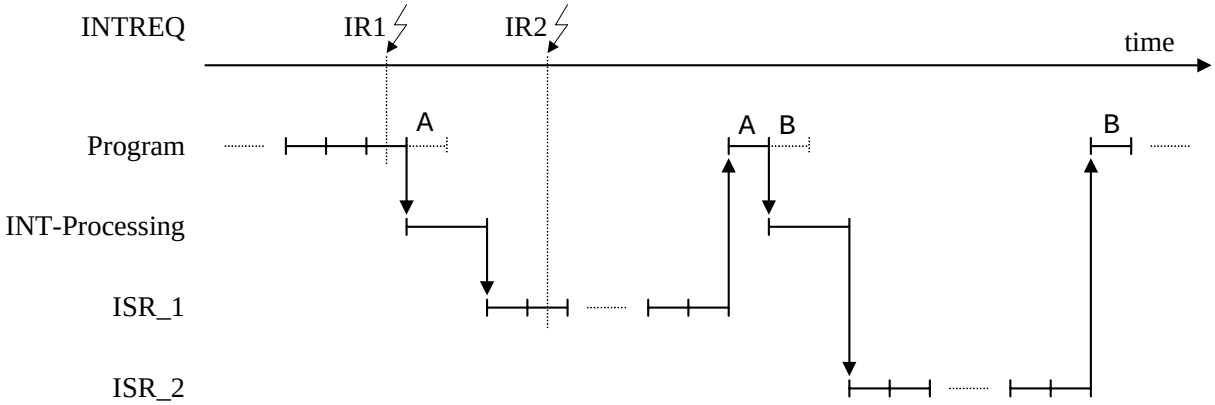
Für die Implementierung des `iret`-Befehls nehmen wir an, dass er abweichend vom “normalen” Operationsprinzip implementiert wurde, und zwar derart, dass nach dem `exec(iret)` keine INTREQ-Abfrage erfolgt.

Dies ist bei CPUs durchaus üblich, da so sichergestellt werden kann, dass selbst bei 100% Interruptauslastung das “Hintergrund-Programm” langsam aber doch weiter ausgeführt wird.

Bei jedem ISR-Wechsel wird mindestens 1 Befehl des “Hintergrund-Programms” abgearbeitet.

3.5.2. Zeitlicher Ablauf (*without nesting*)

Das Programm, die sogenannte „Hintergrundapplikation“, läuft mit PSW.IF=’1’. D.h. ein aktives Signal auf `INTREQ` führt zu einer Unterbrechung.



Die Anforderung von IR1 wird vom *Interrupt-Controller* zur CPU durchgeschaltet, die übermittelte Vektor-Nummer führt zur ISR_1. Während des *Interrupt-Processings* wird der `INTREQ` automatisch deaktiviert (PSW.IF=‘0’). Die IR1-*State-Machine* befindet sich im Zustand `under_service`.

Jetzt tritt die Anforderung IR2 auf. Wir nehmen an, dass IR2 eine Anforderung mit höherer Priorität als IR1 ist und dass der *Interrupt-Controller* höher priorisierte Anforderungen der CPU meldet, obwohl die gerade servicierte, niederpriore Anforderung noch nicht beendet (noch kein EOI) ist. Die IR2-State-Machine bleibt im Zustand `pending` und die `INTREQ`-Leitung bleibt aktiv.

Gegen Ende der ISR_1 wird die entsprechende EOI-Nachricht an den *Interrupt-Controller* gesendet. Die IR1-State-Machine wechselt in den Zustand `idle`. Am Ende der ISR_1 wird der `iret`-Befehle exekutiert, der die CPU zurück ins Programm bringt. Mit dem `iret` wird das PSW wieder hergestellt (PSW.IF=‘1’). Nach der Exekution des Befehls A wird der INTREQ endlich quittiert,- die übermittelte Vektor-Nummer führt zur ISR_2. Die IR2-State-Machine geht in den Zustand `under_service`. Und so weiter ....

3.6. Interrupts mit Verschachtelung

Wenn eine, i.A. höher priorisierte, Interruptanforderung eine gerade laufende Interrupt-Service-Routine unterbrechen darf, dann spricht man von einem System mit verschachtelten Interrupts (*interrupt nesting*). Dieses komplexe Modell erfordert eine sorgsame Planung, da nun ISRs zu einander in paralleler Beziehung stehen. Vom Prinzip her hat man ein *pre-emptives* Multitasking ohne „Software"-Betriebssystem. Der Scheduler des Interrupt-Controllers übernimmt die Rolle des Betriebssystemkerns.

Für kleinste Anwendungen mit wenigen Interruptquellen ist diese Variante sicher pragmatisch und kostengünstig. Bei komplexeren Anwendungen wuchert das System zu einem „Interrupt-Monster" (siehe weiter unten) aus, welches i.A. nicht mehr beherrschbar ist.

3.6.1. Aufbau der ISR (*with nesting*)

Während des *Interrupt-Processings* wird das PSW.IF auf 0 gesetzt. Damit werden alle weiteren Interrupt-Anforderungen von der CPU ignoriert. In der ISR muss nun an geeigneter Stelle das PSW.IF auf 1 gesetzt werden, damit weitere Interruptanforderungen von der CPU bearbeitet werden können.

- Sichern der CPU-Register, etc. (`push`)
- "Ruhigstellen" der Interrupt-Quelle

  damit das IR-Signal wieder inaktiv wird
- `enable-int`
- die eigentliche Funktion der ISR
- Restaurieren der CPU-Register, etc. (`pop`)
- EOI-Nachricht an den Interrupt-Controller
- Abschluss mit `iret`

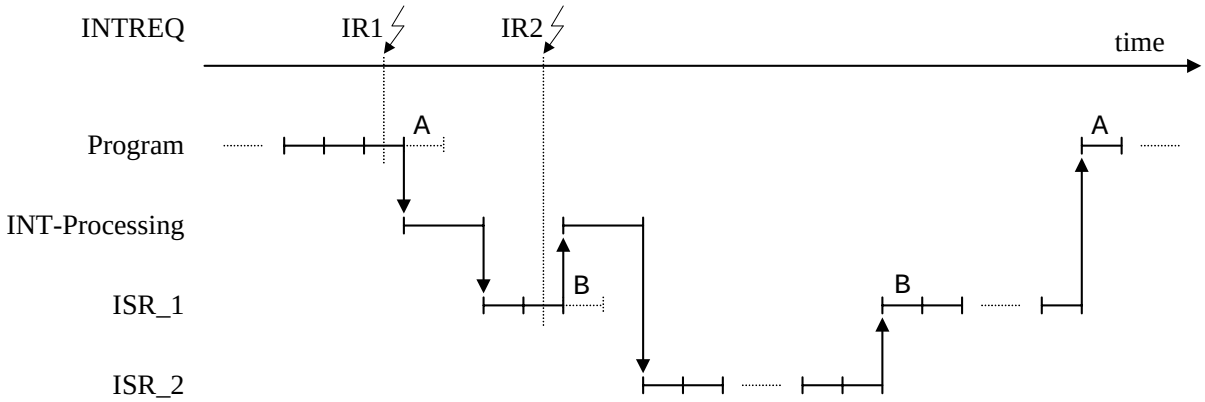
Die zentrale Frage lautet: *„Wann soll die ISR den `INTREQ` wieder freigeben?"*

Im Sinne eines priorisierten, verschachtelten Interrupt-Systems natürlich so früh wie möglich. Sinnvollerweise sollte das "Ruhigstellen" der Interruptquelle, d.h. das Beseitigen der

Interruptursache ohne Störung, d.h. ohne eine weitere Unterbrechung erfolgen. Ob das Sichern der CPU-Register (je nach CPU eine mehr oder weniger zeitaufwendige Tätigkeit) für die weitere Verarbeitung vor oder nach dem `enable-int` erfolgen soll, ist sicher von Fall zu Fall verschieden.

3.6.2. Zeitlicher Ablauf (*with nesting*)

Das Programm, die sogenannte „Hintergrundapplikation“, läuft mit PSW.IF=’1’. D.h. ein aktives Signal auf `INTREQ` führt zu einer Unterbrechung.



Die Anforderung von IR1 wird vom Interrupt-Controller zur CPU durchgeschaltet, die übermittelte Vektor-Nummer führt zur ISR_1. Während des *Interrupt-Processings* wird der `INTREQ` automatisch deaktiviert (PSW.IF=‘0’). Die IR1-*State-Machine* befindet sich im Zustand `under_service`. Innerhalb der ISR_1 wird der `enable-int` Befehl (PSW.IF=‘1’) ausgeführt. Damit führt eine aktive `INTREQ`-Leitung zu einer weiteren Interruptverarbeitung.

Jetzt tritt die Anforderung IR2 auf. Wir nehmen wieder an, dass IR2 eine Anforderung mit höherer Priorität als IR1 ist und dass der Interrupt-Controller höher priore Anforderungen der CPU meldet, obwohl die gerade servicierte, niederpriore Anforderung noch nicht beendet (noch kein EOI) ist. Die IR2-State-Machine geht in den Zustand `pending` und die `INTREQ`-Leitung wird aktiv.

Die CPU akzeptiert den `INTREQ`,- die übermittelte Vektor-Nummer führt zur ISR_2. Die IR2-State-Machine ist im Zustand `under_service`. Der höher priore Interrupt hat nun die niederprioire ISR unterbrochen.

3.7. Definition der Zeiten

Im Zusammenhang mit der Interruptverarbeitung können folgende Zeiten definiert werden:

- *Latency Time*

  vom IR bis zum Einstieg in die ISR

- *Reaction Time*

  vom IR bis zur "Beruhigung" der Quelle

- *Completion Time*

  vom IR bis zum Ende der ISR

Im Datenblatt des Prozessors kann i.A. nur die *Latency Time* spezifiziert werden. Und diese nur unter idealen Vorraussetzungen: der gerade laufende Befehl wird fertig ausgeführt, und dann beginnt sofort der Umstieg auf die parallele Verarbeitungsebene der Interrupt-Service-Routine. Läuft jedoch bereits eine ISR die den INTREQ noch nicht freigegeben hat, dann erhöht sich natürlich die Latenz-Zeit.

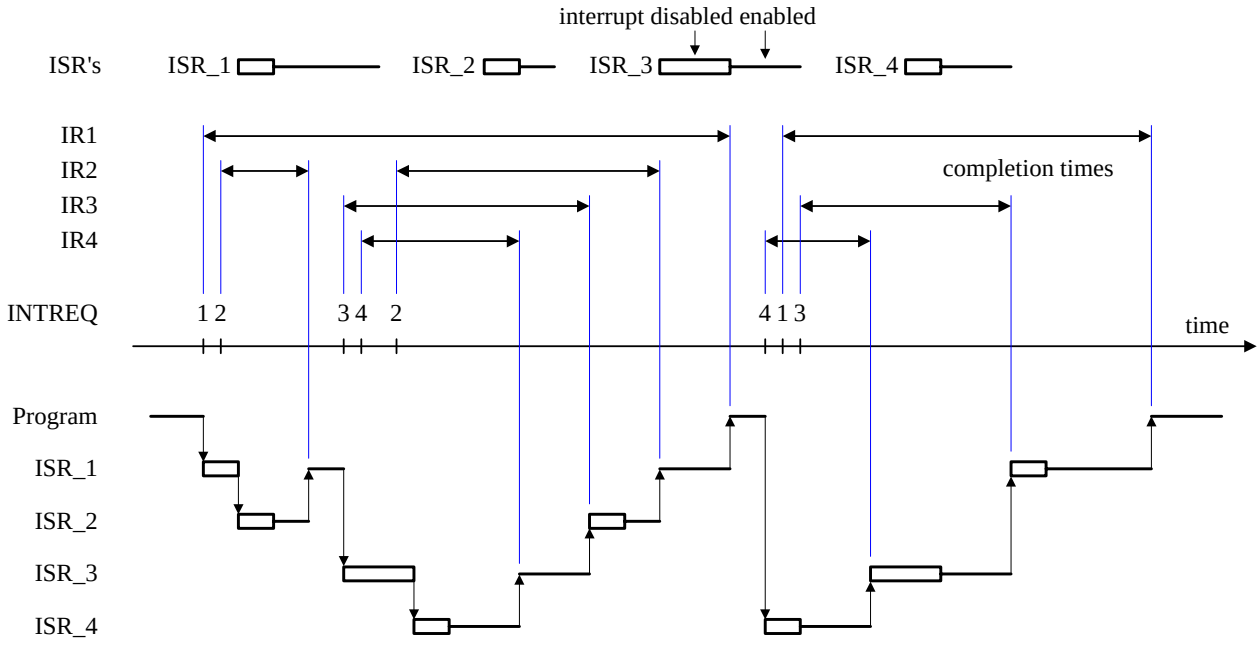
Bei nicht zu komplexen Systemen können nun "*Worst-Case*" Szenarien überlegt werden, aus denen man die Zeiten ermitteln kann. Bei komplexeren Systemen wird das Interruptverhalten mit statistischen Abfolgen von Interruptanforderungen simuliert.

3.8. Das Interrupt-Monster

Das folgende Beispiel zeigt an einem willkürlichen Szenario mit 4 verschachtelten Interrupts, dass das Zeitverhalten sehr schwer vorherzusagen ist. Der Interrupt IR1 hat die niedrigste und IR4 die höchste Priorität. Interessant ist das Verhältnis zwischen der Länge der ISR und der *Completion Time* in den verschiedenen Phasen der Ausführung.

Das folgende Beispiel soll zeigen, dass ein priorisiertes, verschachteltes Interruptsystem ein hohes Maß an zeitlichem Indeterminismus aufweist. Abhängig von der Abfolge von Interrupts, dauert die Ausführung der einzelnen ISRs unterschiedlich lange. Siehe z.B. IR2:

Obwohl die ISR_2 selbst nie von einem höherprioren IR unterbrochen wird, ist die *Completion Time* von IR2 unterschiedlich!



Anmerkung:

> Um die Zeichnung einfacher zu gestalten, wurde angenommen, dass der `iret`-Befehl gemäß der "normalen" Operationsvorschrift implementiert wurde,- d.h. unmittelbar nach dem `exec(iret)` wird eine INTREQ-Abfrage durchgeführt. Liegt eine Anforderung vor, dann wird sofort zur neuen ISR verzweigt.
> Weiters wurde in obiger Zeichnung das *Interrupt-Processing* nicht mehr eingezeichnet,- es liegt implizit in den Übergangspfeilen zwischen den parallelen Verarbeitungsebenen.

Weiters erkennt man, dass die lange Zeit in der INTREQ während der Ausführung von ISR_3 gesperrt ist, die Latenz-Zeit von IR4 (höhere Priorität als IR3!) negativ beeinflusst.

Da die Priorisierung der IR vom externen Interrupt-Controller vorgenommen wird, und der natürlich “keine Ahnung” vom Zustand des Gesamtsystems hat, ist es fast unmöglich ein solches System zeitlich “vorhersagbar” (*predictable*) zu realisieren.

Die “ordnende Instanz” in stark reaktiven Systemen stellt ein *Real-Time-Kernel* (RTK) dar. Ein RTK ist ein “optimiertes” Multitasking-Betriebssystem für effektive Verarbeitung von asynchronen Ereignissen. Die ISRs haben minimale Länge und “beruhigen” nur die Interruptquelle. Danach signalisieren sie das Ereignis dem RTK. Die eigentliche Ereignisverarbeitung wird von Prozessen (Tasks) erledigt,- deren Ausführungsabfolge steuert der RTK abhängig vom Zustand des Gesamtsystems. Die ISRs sind nicht verschachtelt und der Interrupt-Controller “degeneriert” zum “reinen” Interrupt-Multiplexer.

4. Lehrzielorientierte Fragen

1. Was ist eine 1-Adress-Maschine?
2. Erklären Sie die Funktion des *Instruction Address Register*s.
3. Warum muss das IAR vor der Befehlsausführung erhöht werden?
4. Was ist das *Program Status Word*?
5. Welche Arten von Flags enthält das PSW?
6. Was ist eine bedingte Verzweigung?
7. Was ist der *Stack*?
8. Was ist der *Stack Pointer*?
9. Was versteht man unter einem *Stack-Frame*?
10. Welche CPU-Ressourcen benötigt man, um *Stack-Frames* zu verwalten?
11. Was unterscheidet einen Sprung- von einem Unterprogrammaufruf-Befehl?
12. Was macht der `return`-Befehl?

13. Was wird durch einen *Hardware-Interrupt* erreicht?
14. Was motiviert den *Hardware-Interrupt*?
15. Was versteht man unter *Polling*?
16. Wozu benötigt man das *Interrupt-Flag*?
17. Was versteht man unter der „Transparenz“ der *Interrupt Service Routine*?
18. Wie sieht der prinzipielle Aufbau einer ISR aus?
19. Was versteht man unter dem Begriff „*Interrupt-Processing*“?
20. Warum kann der `iret`-Befehl i.A. nicht „ausprogrammiert“ werden?
21. Kann die Ausführung eines Befehls durch einen Interrupt unterbrochen werden?

22. Was versteht man unter dem vektoriellen Interruptkonzept?
23. Wie findet beim vektoriellen Interruptkonzept die CPU zur passenden Interrupt-Service-Routine?

24. Welche Funktionen hat ein Interrupt-Controller?
25. Wozu benötigt man die Interrupt Vektor Nummer?
26. Wann wird die Interrupt Vektor Nummer an die CPU übertragen?
27. Was versteht man unter *Interrupt Nesting*?
28. Wie kann der `iret` Befehl implementiert sein?
29. Wie kann die Operation zum Lesen der Interrupt Vektor Nummer implementiert werden?
30. Wie ist die Latenzzeit definiert?
31. Was versteht man unter der *Completion Time*?
32. Sind ISRs parallel zu einander?
33. Sind ISRs parallel zum „Hintergrundprogramm“?
34. Erklären Sie die IR-State-Machine für einen Interrupt-Request Eingang eines Interrupt Controllers.