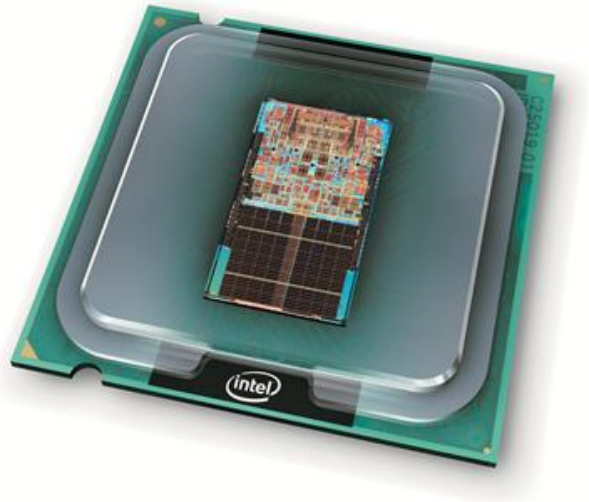


Bare Metal Development

Alejandro Furfaro

8 de abril de 2020

Temario

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro `eax` en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **eax** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **`eax`** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **eax** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **eax** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **`eax`** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **`eax`** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

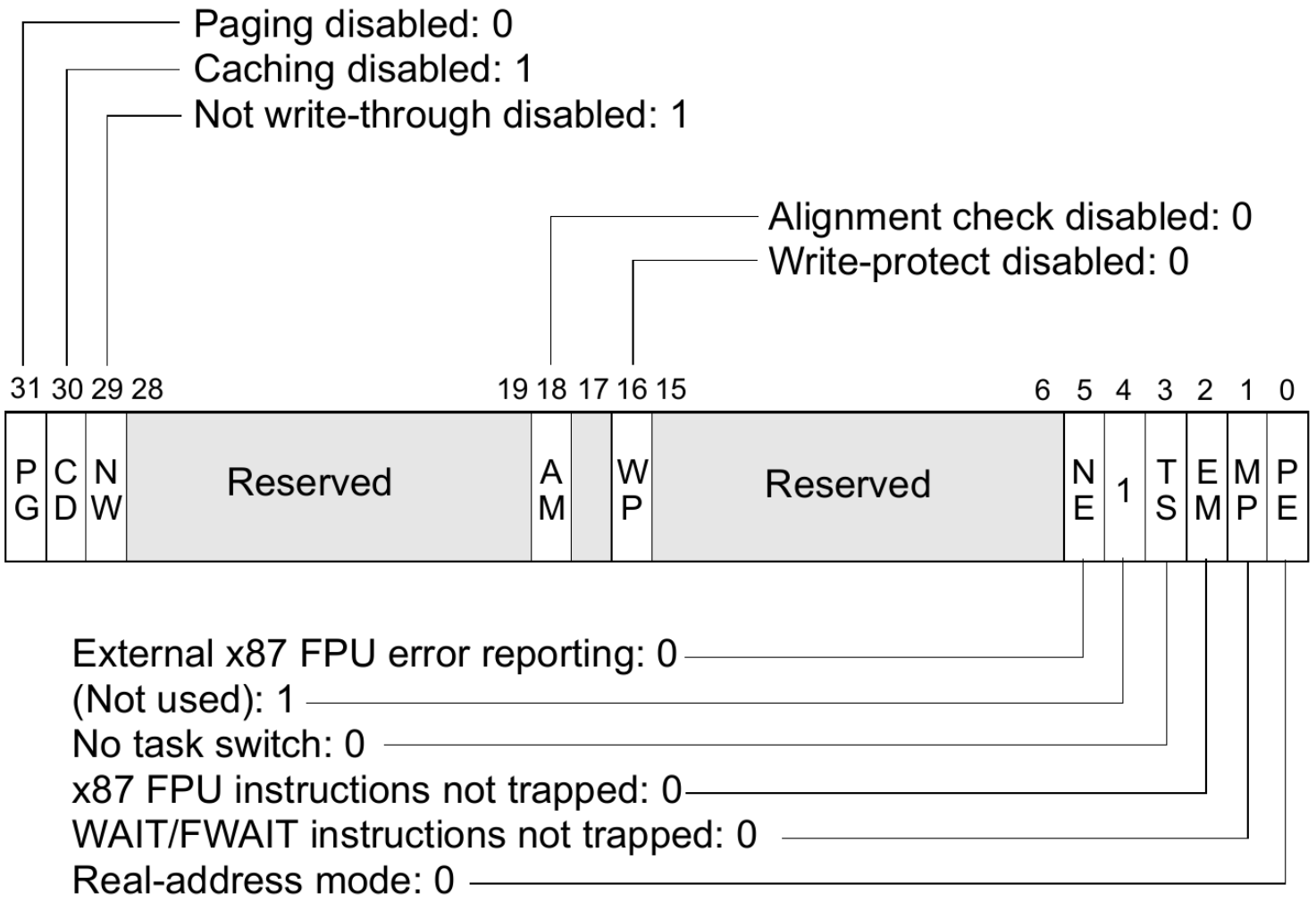
1. Built In Self Test

- Es un test interno que ejecuta el procesador luego de su encendido o cuando se activa su pin #RESET
- Como resultado inicializa el registro **`eax`** en 0x0 solo si todos los test pasaron OK.
- El resto de los registros toman un valor bien determinado
- Se invalidan las memorias cache internas
- Se invalidan las TLB[1]
- Se limpian los Branch Target Buffers[2]
- La diferencia con enviar una señal #INIT es que en éste caso, se mantienen intactos los valores de los Registros de la FPU, los MSR's y los MTRR's, y los caches internos.
- De acuerdo a la familia de procesador implementa con variantes la inicialización de los demás procesadores presentes en caso de un sistema SMP

[1] Ver Paginación de Memoria

[2] Ver Microarquitectura - Predicción de saltos

2. Valores iniciales de interés



Registro CR0. Valor inicial

2. Valores iniciales de interés

Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

Valor iniciales de los registros corrientes[3]

[3] Para ampliar: Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a. Capítulo 9

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector

- Del slide anterior nos quedamos con los registros CS:IP, cuyos valores iniciales son respectivamente 0xF000:0xFFF0.
- Al mismo tiempo el procesador está en modo real de modo que su rango de direccionamiento físico es desde 0x00000 hasta 0xFFFFF, es decir 1Mbyte.
- de modo que la dirección física que generaría un procesador 8086 viene dada por la expresión: $cs << 4 + ip$
- Esto llevará al procesador a realizar el primer OPCODE FETCH en la dirección física 0xFFFF0.
- En este rango de direcciones es necesario mapear una memoria No volátil.

3. Reset Vector (Problemas asociados al legacy)

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aún la moderna 10^{ma} generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente.
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Intel pensó por lo tanto en una alternativa

3. Reset Vector (Problemas asociados al legacy)

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aún la moderna 10^{ma} generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente.
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Intel pensó por lo tanto en una alternativa

3. Reset Vector (Problemas asociados al legacy)

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aún la moderna 10^{ma} generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente.
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Intel pensó por lo tanto en una alternativa

3. Reset Vector (Problemas asociados al legacy)

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aún la moderna 10^{ma} generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente.
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Intel pensó por lo tanto en una alternativa

3. Reset Vector (Problemas asociados al legacy)

- Uno de nuestros principales problemas consiste en que el 8086 hace tiempo que no existe y en cambio sus descendientes compatibles poseen un rango de direccionamiento muchísimo mas amplio.
- Sin embargo por compatibilidad los procesadores posteriores (aún la moderna 10^{ma} generación Core) arrancan en Modo Real accediendo en principio al espacio de direccionamiento de 1Mbyte que referimos anteriormente.
- Por lo tanto, necesitaríamos colocar una memoria no volátil alrededor del primer Mbyte, interrumpiendo la continuidad de los grandes bancos de DRAM que desde hace tiempo se comercializan.
- Intel pensó por lo tanto en una alternativa

3. Reset Vector (Recursos de MP en MR)

- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real
- Y por lo tanto les asigna valores que resulten convenientes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.
- Por lo tanto en el reset el estado de estos registros “amanece”como vemos a continuación:

3. Reset Vector (Recursos de MP en MR)

- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real
- Y por lo tanto les asigna valores que resulten convenientes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.
- Por lo tanto en el reset el estado de estos registros “amanece”como vemos a continuación:

3. Reset Vector (Recursos de MP en MR)

- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real
- Y por lo tanto les asigna valores que resulten convenientes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.
- Por lo tanto en el reset el estado de estos registros “amanece”como vemos a continuación:

3. Reset Vector (Recursos de MP en MR)

- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real
- Y por lo tanto les asigna valores que resulten convenientes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.
- Por lo tanto en el reset el estado de estos registros “amanece”como vemos a continuación:

3. Reset Vector (Recursos de MP en MR)

- Los registros de segmento en Modo Protegido trabajan con un registro cache para guardar sus descriptores a fin de evitar acceder a las tablas de descriptores cada vez que se necesita efectuar un acceso a memoria (o sea... siempre!).
- Todo lo que hacen los procesadores posteriores al 8086 es usarlos en modo real
- Y por lo tanto les asigna valores que resulten convenientes, para diseñar el mapa de memoria de un sistema x86 de manera mas sensata.
- Por lo tanto en el reset el estado de estos registros “amanece”como vemos a continuación:

3. Reset Vector - Recursos de MP en MR

		Base Address	Limit	Attrib
CS	0xF000	0xFFFF0000	0x0000FFFF	XX
DS	0x0000	0x00000000	0x0000FFFF	XX
ES	0x0000	0x00000000	0x0000FFFF	XX
SS	0x0000	0x00000000	0x0000FFFF	XX
FS	0x0000	0x00000000	0x0000FFFF	XX
GS	0x0000	0x00000000	0x0000FFFF	XX

Sumando a esto que el registro IP en el reset vale 0xFFF0, la dirección en la que el procesador aún en modo real irá a buscar su primer instrucción se resuelve mediante la siguiente expresión:

```
CS.BaseAddress + IP. = 0xFFFFFFF0
```

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

3. Reset Vector - Mapa de memoria inicial

- El procesador inicia su operación buscando operaciones en el fondo de la memoria. Puntualmente a 16 bytes de los 4Gbytes.
- Esto coincide con el anterior comportamiento de iniciar a 16 bytes del Mbyte del 8086.
- Es decir que el procesador inicia en Modo real con su restricción de espacio de direccionamiento a 1 Mbyte de memoria, pero puede realizar opcode fetch muy por encima de esa dirección. . . raro. . .
- Hay una condición para esto:

La letra chica del contrato :)

El valor del registro CS ***no puede cambiar mientras el procesador trabaje en modo real***. Ni bien lo haga, la Base Address de su registro cache se pone en 0x00000000, y automáticamente queda restringido a continuar buscando sus instrucciones en el primer Mega Byte de memoria.

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendable para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección de Modo

- En este punto es conveniente ocuparse de establecer el modo en que el procesador debe funcionar.
- Hay tres posibilidades.
  1. Dejarlo en modo Real
  2. Establecer Modo Protegido Flat
  3. Establecer Modo Protegido Segmentado (no recomendado para escribir firmware)
- Los modos 2 y 3 son variantes del Modo Protegido y tiene que ver con la forma en que organizaremos el modelo de segmentación

4. Selección del modo

- En la práctica la ROM se mapea físicamente en el fondo de los 4 Gbytes.
- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeño que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activará el la línea Chip Select de este componente será: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construirá un bloque de 4 Kbytes de código con una visión de un mapa de memoria de 1 Mbyte de capacidad y lo ubicará en los últimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicará en 0xFFFFF000.

4. Selección del modo

- En la práctica la ROM se mapea físicamente en el fondo de los 4 Gbytes.
- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeño que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activará el la línea Chip Select de este componente será: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construirá un bloque de 4 Kbytes de código con una visión de un mapa de memoria de 1 Mbyte de capacidad y lo ubicará en los últimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicará en 0xFFFFF000.

4. Selección del modo

- En la práctica la ROM se mapea físicamente en el fondo de los 4 Gbytes.
- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeño que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activará el la línea Chip Select de este componente será: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construirá un bloque de 4 Kbytes de código con una visión de un mapa de memoria de 1 Mbyte de capacidad y lo ubicará en los últimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicará en 0xFFFFF000.

4. Selección del modo

- En la práctica la ROM se mapea físicamente en el fondo de los 4 Gbytes.
- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeño que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activará el la línea Chip Select de este componente será: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construirá un bloque de 4 Kbytes de código con una visión de un mapa de memoria de 1 Mbyte de capacidad y lo ubicará en los últimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicará en 0xFFFFF000.

4. Selección del modo

- En la práctica la ROM se mapea físicamente en el fondo de los 4 Gbytes.
- Por ejemplo: Si usamos una EEPROM de 4 Kbytes (lo mas pequeño que podemos conseguir actualmente en el mercado), el rango de direcciones para las que se activará el la línea Chip Select de este componente será: 0xFFFFF000 - 0xFFFFFFFF.
- Sin embargo cuando empecemos a escribir el programa le vamos a indicar al ensamblador que trabajamos en modo real y por lo tanto en el rango de direcciones 0xFF000-0xFFFFF.
- El ensamblador solo construirá un bloque de 4 Kbytes de código con una visión de un mapa de memoria de 1 Mbyte de capacidad y lo ubicará en los últimos 4 Kbytes de ese espacio. El decodificador de memoria de hardware lo ubicará en 0xFFFFF000.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.

- El único código para iniciar es un loop infinito.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.
- El único código para iniciar es un loop infinito.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.
- El único código para iniciar es un loop infinito.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.

```
ORG 0xFF000  ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16        ;Indica al ensamblador generar código de 16 bits
```

- El único código para iniciar es un loop infinito.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.

```
ORG 0xFF000  ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16        ;Indica al ensamblador generar código de 16 bits
```

- El único código para iniciar es un loop infinito.

5. Primer imagen de 4K

- En primer lugar hay que ponerle un marco de trabajo al ensamba-dor.
- No se genera código. Simplemente son directivas para el ensam-blador.

```
ORG 0xFF000 ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16       ;Indica al ensamblador generar código de 16 bits
```

- El único código para iniciar es un loop infinito.

```
init16:
      cli         ;Deshabilita interrupciones
      jmp init16  ;loop infinito
align 16          ;Completa hasta la siguiente dirección múltiplo
                  ;de 16 (verificar con que completa con NOP's).
```

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

5.1. Completando la imagen de 4Kbytes

- Necesitamos que el label **`init16`** sea mapeado por el ensamblador exactamente en la dirección del reset vector: **`0xFFFF0`**.
- la sentencia **`align 16`** vista solo nos asegura que el código generado a partir de **`init16`**, se complete con **`NOP`**'s hasta la dirección previa a la siguiente dirección alineada a 16 bits. Es decir, nos asegura alinear el final de los 4Kbytes.
- Necesitamos completar los primeros 4080 bytes de los 4096 totales ya que de otro modo **`init16`** estará al principio de los 4K.
- Primer impulso, que ocupe efectivamente la dirección `0xFFFF0`:

```
    TIMES 4080 db 0x90   ;Generamos 4080 NOP's
init16:
      cli            ;Deshabilita interrupciones
      jmp init16     ;loop infinito
align 16             ;Completa hasta la siguiente dirección múltiplo
                     ;de 16 (verificar con que completa con NOP's).
```

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.1 Completando la imagen de 4Kbytes

- El código anterior resuelve el problema pero no es escalable.
- A medida que agreguemos código para hacer algo mínimamente útil, deberemos ajustar el parámetro de TIMES para reducir de 4080 a la cantidad que haga falta de acuerdo con el tamaño del código que incluyamos.
- En realidad lo mas complejo será llevar la cuenta del tamaño del código.
- Por lo tanto hay que buscar un método mas eficiente.
- En el siguiente código EQU (por Equal) le genera una constante al ensamblador cuyo valor calcula automáticamente el tamaño del código generado y lo resta del tamaño de la ROM. Problema resuelto.

5.2. Versión final de nuestro primer intento

```
ORG 0xFF000 ;Esto es: (1MB-4KB) -> 0x100000-0x1000=0xFF000.
USE16       ;Indica al ensamblador generar código de 16 bits

code_size EQU (end - init16) ; siempre es el tamaño del código

; Rellenamos la ROM con 0x90 (NOP)
times (4096-code_size) db 0x90

init16:
    cli         ;Deshabilita interrupciones
    jmp init16  ;loop infinito
aqui:
    hlt         ;Si por algún motivo sale del loop: HALT
    jmp aqui    ;Solo sale por reset o por interrupción
align 16        ;Completa hasta la siguiente dirección múltiplo
                ;de 16 (verificar con que completa con NOP's).
end:
```

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[2] (opcional).
  2. Inicializar el soporte del procesador para manejo de memoria.
  3. Inicializar el chipset[3].

[2] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol 3a.
[3] BIOS/Firmware Writer's Guide (a Black Magic Manu

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[4] (opcional)
  2. Inicializar el soporte del procesador para manejo de memoria[5]
  3. Inicializar el chipset[5]

[4] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a.

[5] BIOS/Firmware Writer's Guide (a Black Magic Manual)

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[4] (opcional)
  2. Inicializar el soporte del procesador para manejo de memoria[5]
  3. Inicializar el chipset[5]

[4] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a.

[5] BIOS/Firmware Writer's Guide (a Black Magic Manual)

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[4] (opcional)
  2. Inicializar el soporte del procesador para manejo de memoria[5]
  3. Inicializar el chipset[5]

[4] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a.

[5] BIOS/Firmware Writer's Guide (a Black Magic Manual)

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[4] (opcional)
  2. Inicializar el soporte del procesador para manejo de memoria[5]
  3. Inicializar el chipset[5]

[4] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a.

[5] BIOS/Firmware Writer's Guide (a Black Magic Manual)

6. Preparar la Inicialización de memoria

- Todo el código que se ha ejecutado hasta este momento, se ha ejecutado desde NVRAM o sea Flash.
- Cuanto antes pasemos a ejecutar desde memoria DRAM, antes aceleraremos el inicio del sistema ya que la memoria DRAM es mas veloz que la Flash.
- Entonces es urgente inicializar la DRAM. Para ello hay que:
  1. Actualizar el microcódigo del procesador[4] (opcional)
  2. Inicializar el soporte del procesador para manejo de memoria[5]
  3. Inicializar el chipset[5]

[4] Ampliar desde Sección 9.11 Microcode Update Facilities, Intel®64 and IA-32 Architectures Software Developer's Manual. Vol 3a.

[5] BIOS/Firmware Writer's Guide (a Black Magic Manual)

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory Initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory Initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

6. Inicialización de Memoria

- Otra caja negra.
- Intel provee (NDA mediante) los siguientes recursos:
  1. BIOS/Firmware Writer's Guide
  2. Memory initialization Reference Code (MRC)
- Es chipset (plataforma) dependiente
- MRC puede estar escrito para ejecutar en código de 16 bits o 32 bits (Segmentación Flat o multisegmento).

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re Direction
  4. Configurar un Stack
  5. Transferir control de progrema a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7. Inicialización Post Memoria

- Una vez ejecutado el código que inicializa el controlador de memoria y los configura de acuerdo a los DIMMs detectados en el sistema,se termina el trabajo de inicialización de memoria.
- Es una secuencia de pasos adicional, pero hecha por nosotros
  1. Memory test
  2. Shadow Firmware
  3. Memory Transaction Re-Direction
  4. Configurar un Stack
  5. Transferir control de programa a DRAM

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.1. Memory Test

- Es un primer uso de la memoria para comprobar su integridad una vez inicializado el controlador lo cual nos permite el acceso
- Puede estar incluido en el MRC (algunos vendors lo proveen)
- Es conveniente chequear memoria en este punto ya que los errores se producen de manera aleatoria e inconsistente.
- Simplifica el debug del firmware ya que si pasó el test, los errores posteriores no son atribuibles a memoria.
- Los test son un balance entre la granularidad (cada cuantos bytes se comprueba) vs. performance (en móviles el tiempo de arranque es importante)
- No hay algoritmos específicos, sino que son mas bien simples, y pueden hacerse en assembler.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Firmware Shadow

- Es código que se auto copia desde NVRAM (Lenta, típicamente es FLASH) a DRAM (mas rápida y por lo tanto mejora el tiempo del setup).
- Al habilitar el cache, cada acceso a la Flash (NVRAM) genera un Miss y retrasa enormemente la ejecución.
- Copiarlo a DRAM no solo acelera por ser esta mas rápida que la NVRAM, sino por ser “cacheable”.
- En los sistemas PC la zona de shadow debe copiarse desde la NVRAM hasta la zona por debajo del Mbyte.
- En otros sistemas el espacio de direccionamiento destino de la copia es arbitraria.

7.2. Memory Transaction Re-Direction

- Los chipsets generalmente están provistos de Registros PAM (Programmable Attribute Maps)
- Estos PAMs permiten controlar el aliasing de direcciones que hace posible leer o escribir secciones de memoria por debajo de 1 Mbyte hacia o desde DRAM o NVRAM cuyas direcciones rondan los 4 Gbytes.
- Antes de ejecutar shadowing puede requerirse acceder a estos registros.
- Los accesos dependen de cada chipset. Algunos permiten leer y escribir, otros solo leer.

7.2. Memory Transaction Re-Direction

- Los chipsets generalmente están provistos de Registros PAM (Programmable Attribute Maps)
- Estos PAMs permiten controlar el aliasing de direcciones que hace posible leer o escribir secciones de memoria por debajo de 1 Mbyte hacia o desde DRAM o NVRAM cuyas direcciones rondan los 4 Gbytes.
- Antes de ejecutar shadowing puede requerirse acceder a estos registros.
- Los accesos dependen de cada chipset. Algunos permiten leer y escribir, otros solo leer.

7.2. Memory Transaction Re-Direction

- Los chipsets generalmente están provistos de Registros PAM (Programmable Attribute Maps)
- Estos PAMs permiten controlar el aliasing de direcciones que hace posible leer o escribir secciones de memoria por debajo de 1 Mbyte hacia o desde DRAM o NVRAM cuyas direcciones rondan los 4 Gbytes.
- Antes de ejecutar shadowing puede requerirse acceder a estos registros.
- Los accesos dependen de cada chipset. Algunos permiten leer y escribir, otros solo leer.

7.2. Memory Transaction Re-Direction

- Los chipsets generalmente están provistos de Registros PAM (Programmable Attribute Maps)
- Estos PAMs permiten controlar el aliasing de direcciones que hace posible leer o escribir secciones de memoria por debajo de 1 Mbyte hacia o desde DRAM o NVRAM cuyas direcciones rondan los 4 Gbytes.
- Antes de ejecutar shadowing puede requerirse acceder a estos registros.
- Los accesos dependen de cada chipset. Algunos permiten leer y escribir, otros solo leer.

7.2. Memory Transaction Re-Direction

- Los chipsets generalmente están provistos de Registros PAM (Programmable Attribute Maps)
- Estos PAMs permiten controlar el aliasing de direcciones que hace posible leer o escribir secciones de memoria por debajo de 1 Mbyte hacia o desde DRAM o NVRAM cuyas direcciones rondan los 4 Gbytes.
- Antes de ejecutar shadowing puede requerirse acceder a estos registros.
- Los accesos dependen de cada chipset. Algunos permiten leer y escribir, otros solo leer.

7.3. Establecimiento de un stack

- Antes de saltar a memoria es necesario definir un stack
- Es necesario definir la cantidad de memoria y considerar que el stack crece hacia direcciones bajas.
- En modo 16 bits se configuran los registros `ss:sp`
- En modo Protegido (ya sea Flat o Segmentado) se configuran `ss:esp`

7.3. Establecimiento de un stack

- Antes de saltar a memoria es necesario definir un stack
- Es necesario definir la cantidad de memoria y considerar que el stack crece hacia direcciones bajas.
- En modo 16 bits se configuran los registros `ss:sp`
- En modo Protegido (ya sea Flat o Segmentado) se configuran `ss:esp`

7.3. Establecimiento de un stack

- Antes de saltar a memoria es necesario definir un stack
- Es necesario definir la cantidad de memoria y considerar que el stack crece hacia direcciones bajas.
- En modo 16 bits se configuran los registros `ss:sp`
- En modo Protegido (ya sea Flat o Segmentado) se configuran `ss:esp`

7.3. Establecimiento de un stack

- Antes de saltar a memoria es necesario definir un stack
- Es necesario definir la cantidad de memoria y considerar que el stack crece hacia direcciones bajas.
- En modo 16 bits se configuran los registros **`ss:sp`**
- En modo Protegido (ya sea Flat o Segmentado) se configuran `ss:esp`

7.3. Establecimiento de un stack

- Antes de saltar a memoria es necesario definir un stack
- Es necesario definir la cantidad de memoria y considerar que el stack crece hacia direcciones bajas.
- En modo 16 bits se configuran los registros **`ss:sp`**
- En modo Protegido (ya sea Flat o Segmentado) se configuran **`ss:esp`**

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

7.4. Salto a memoria DRAM

- Aquí es donde se puede producir un problema si no se hicieron bien las cosas.
- El programa debe haberse auto copiado de NVRAM a DRAM
- El salto se implementa mediante un salto FAR.
- En modo 16 bits o real hay que saltar a una dirección dentro del primer Mbyte de DRAM.
- En modo Protegido (ya sea Flat o Segmentado) no hay restricciones respecto de la dirección de destino, dentro de los 4 Gbytes de capacidad de direccionamiento. Pero debe estar el sistema correctamente inicializado (tablas de descriptores de segmento mínimas y demás delicias de la vida).

8. Habilitaciones de dispositivos misceláneos

- Este ítem es plataforma dependiente y se compone de un conjunto de dispositivos que surgen del análisis de los esquemáticos de hardware.
- Los típicos dispositivos que están presentes en todos los chipsets (con las variantes de implementación de cada caso) son:
  1. Programación del chip de reloj.
  2. GPIO. Igualmente debe consultarse el Current Bios Writer Guide (NDA mediante) para tener los detalles.

8. Habilitaciones de dispositivos misceláneos

- Este ítem es plataforma dependiente y se compone de un conjunto de dispositivos que surgen del análisis de los esquemáticos de hardware.
- Los típicos dispositivos que están presentes en todos los chipsets (con las variantes de implementación de cada caso) son:
  1. Programación del chip de reloj
  2. GPIO: Igualmente debe consultarse el Current Bios Writer (NDA mediante) para tener los detalles

8. Habilitaciones de dispositivos misceláneos

- Este ítem es plataforma dependiente y se compone de un conjunto de dispositivos que surgen del análisis de los esquemáticos de hardware.
- Los típicos dispositivos que están presentes en todos los chipsets (con las variantes de implementación de cada caso) son:
  1. Programación del chip de reloj
  2. GPIO. Igualmente debe consultarse el Chipset BIOS Writer Guide (NDA mediante) para tener los detalles.

8. Habilitaciones de dispositivos misceláneos

- Este ítem es plataforma dependiente y se compone de un conjunto de dispositivos que surgen del análisis de los esquemáticos de hardware.
- Los típicos dispositivos que están presentes en todos los chipsets (con las variantes de implementación de cada caso) son:
  1. Programación del chip de reloj
  2. GPIO. Igualmente debe consultarse el Chipset BIOS Writer Guide (NDA mediante) para tener los detalles.

8. Habilitaciones de dispositivos misceláneos

- Este ítem es plataforma dependiente y se compone de un conjunto de dispositivos que surgen del análisis de los esquemáticos de hardware.
- Los típicos dispositivos que están presentes en todos los chipsets (con las variantes de implementación de cada caso) son:
  1. Programación del chip de reloj
  2. GPIO. Igualmente debe consultarse el Chipset BIOS Writer Guide (NDA mediante) para tener los detalles.

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:
  1. Programmable Interrupt Controller (PIC)
  2. Local Advanced Programmable Interrupt Controller (APIC)
  3. Input/Output Advanced Programmable Interrupt Controller (I/OAPIC)
  4. Messaged Signaled Interrupt (MSI)

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:

1. Programmable Interrupt Controller (PIC)
2. Local Advanced Programmable Interrupt Controller (APIC)
3. Input /Output Advanced Programmable Interrupt Controller (IOxAPIC)
4. Messaged Signaled Interrupt (MSI)

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:

1. Programmable Interrupt Controller (PIC)
2. Local Advanced Programmable Interrupt Controller (APIC)
3. Input /Output Advanced Programmable Interrupt Controller (IOxAPIC)
4. Messaged Signaled Interrupt (MSI)

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:

1. Programmable Interrupt Controller (PIC)
2. Local Advanced Programmable Interrupt Controller (APIC)
3. Input /Output Advanced Programmable Interrupt Controller (IOxAPIC)
4. Messaged Signaled Interrupt (MSI)

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:

1. Programmable Interrupt Controller (PIC)
2. Local Advanced Programmable Interrupt Controller (APIC)
3. Input /Output Advanced Programmable Interrupt Controller (IOxAPIC)
4. Messaged Signaled Interrupt (MSI)

9. Habilitaciones de Interrupciones

- Los procesadores de Intel pueden manejar las interrupciones mediante cualquiera de los siguientes métodos o combinación de ellos:

1. Programmable Interrupt Controller (PIC)
2. Local Advanced Programmable Interrupt Controller (APIC)
3. Input /Output Advanced Programmable Interrupt Controller (IOxAPIC)
4. Messaged Signaled Interrupt (MSI)

Es hora de cargar un Sistema Operativo

¿Que sabemos del linker?

- Combina diferentes objetos reubicables y objetos encapsulados en bibliotecas soportados por **BFD** (***B**inary **F**ile **D**escriptor*), en un archivo de salida.
- **BFD** es un acuerdo mediante el cual cada objeto tiene un encabezado en el cual contiene información descriptiva, una cantidad de secciones (variable),cada una con su nombre, atributos y bloque de datos (o código), una tabla de símbolos, entradas de relocación, entre las partes mas importantes.
- Con todos estos elementos del encabezado el Linker resuelve referencias entre archivos objetos, combina las secciones del mismo tipo y construye un archivo que contiene el código y los datos del programa ejecutable mas un encabezado con la información que el sistema operativo necesita para construir su imagen en memoria para poder ejecutarlo.

¿Que sabemos del linker?

- Combina diferentes objetos reubicables y objetos encapsulados en bibliotecas soportados por **BFD** (***B**inary **F**ile **D**escriptor*), en un archivo de salida.
- **BFD** es un acuerdo mediante el cual cada objeto tiene un encabezado en el cual contiene información descriptiva, una cantidad de secciones (variable),cada una con su nombre, atributos y bloque de datos (o código), una tabla de símbolos, entradas de relocación, entre las partes mas importantes.
- Con todos estos elementos del encabezado el Linker resuelve referencias entre archivos objetos, combina las secciones del mismo tipo y construye un archivo que contiene el código y los datos del programa ejecutable mas un encabezado con la información que el sistema operativo necesita para construir su imagen en memoria para poder ejecutarlo.

¿Que sabemos del linker?

- Combina diferentes objetos reubicables y objetos encapsulados en bibliotecas soportados por **BFD** (***B**inary **F**ile **D**escriptor*), en un archivo de salida.
- **BFD** es un acuerdo mediante el cual cada objeto tiene un encabezado en el cual contiene información descriptiva, una cantidad de secciones (variable),cada una con su nombre, atributos y bloque de datos (o código), una tabla de símbolos, entradas de relocación, entre las partes mas importantes.
- Con todos estos elementos del encabezado el Linker resuelve referencias entre archivos objetos, combina las secciones del mismo tipo y construye un archivo que contiene el código y los datos del programa ejecutable mas un encabezado con la información que el sistema operativo necesita para construir su imagen en memoria para poder ejecutarlo.

¿Que sabemos del linker?

- Combina diferentes objetos reubicables y objetos encapsulados en bibliotecas soportados por **BFD** (***B**inary **F**ile **D**escriptor*), en un archivo de salida.
- **BFD** es un acuerdo mediante el cual cada objeto tiene un encabezado en el cual contiene información descriptiva, una cantidad de secciones (variable),cada una con su nombre, atributos y bloque de datos (o código), una tabla de símbolos, entradas de relocación, entre las partes mas importantes.
- Con todos estos elementos del encabezado el Linker resuelve referencias entre archivos objetos, combina las secciones del mismo tipo y construye un archivo que contiene el código y los datos del programa ejecutable mas un encabezado con la información que el sistema operativo necesita para construir su imagen en memoria para poder ejecutarlo.

¿Que es un *linker script*?

- Es un texto escrito que contiene la secuencia de pasos (script) que describen al linker, como mínimo, en que forma mapear las secciones de los archivos de entrada en el archivo de salida, controlado el layout en memoria de éste último archivo.
- El script puede por supuesto agregar otras actividades, que veremos a lo largo de esta clase. Pero como mínimo cumple con esto.
- Para lograrlo es necesario que la secuencia de pasos esté escrita en un lenguaje entendible por el linker.
- El script se escribe en un archivo y se le especifica al linker mediante la opción $-T$

¿Que es un *linker script*?

- Es un texto escrito que contiene la secuencia de pasos (script) que describen al linker, como mínimo, en que forma mapear las secciones de los archivos de entrada en el archivo de salida, controlado el layout en memoria de éste último archivo.
- El script puede por supuesto agregar otras actividades, que veremos a lo largo de esta clase. Pero como mínimo cumple con esto.
- Para lograrlo es necesario que la secuencia de pasos esté escrita en un lenguaje entendible por el linker.
- El script se escribe en un archivo y se le especifica al linker mediante la opción $-T$

¿Que es un *linker script*?

- Es un texto escrito que contiene la secuencia de pasos (script) que describen al linker, como mínimo, en que forma mapear las secciones de los archivos de entrada en el archivo de salida, controlado el layout en memoria de éste último archivo.
- El script puede por supuesto agregar otras actividades, que veremos a lo largo de esta clase. Pero como mínimo cumple con esto.
- Para lograrlo es necesario que la secuencia de pasos esté escrita en un lenguaje entendible por el linker.
- El script se escribe en un archivo y se le especifica al linker mediante la opción $-T$

¿Que es un *linker script*?

- Es un texto escrito que contiene la secuencia de pasos (script) que describen al linker, como mínimo, en que forma mapear las secciones de los archivos de entrada en el archivo de salida, controlado el layout en memoria de éste último archivo.
- El script puede por supuesto agregar otras actividades, que veremos a lo largo de esta clase. Pero como mínimo cumple con esto.
- Para lograrlo es necesario que la secuencia de pasos esté escrita en un lenguaje entendible por el linker.
- El script se escribe en un archivo y se le especifica al linker mediante la opción $-T$

¿Que es un *linker script*?

- Es un texto escrito que contiene la secuencia de pasos (script) que describen al linker, como mínimo, en que forma mapear las secciones de los archivos de entrada en el archivo de salida, controlado el layout en memoria de éste último archivo.
- El script puede por supuesto agregar otras actividades, que veremos a lo largo de esta clase. Pero como mínimo cumple con esto.
- Para lograrlo es necesario que la secuencia de pasos esté escrita en un lenguaje entendible por el linker.
- El script se escribe en un archivo y se le especifica al linker mediante la opción **`-T`**

¿Y cuando no especificamos un *linker script*?

- Siempre se ejecuta un *linker script*. Por default *ld* tiene uno embebido en su código que se puede ver tipeando

```
ld --verbose
```

Puede ser corroborado dentro del propio archivo /usr/bin/ld, ejecutando

```
hexdump -C /usr/bin/ld |more
```

y navegando con paciencia a lo largo del código hasta encontrar la sección de datos en la que está escrito el script (por eso la opción *-C*)

¿Y cuando no especificamos un *linker script*?

- Siempre se ejecuta un *linker script*. Por default **`ld`** tiene uno embebido en su código que se puede ver tipeando

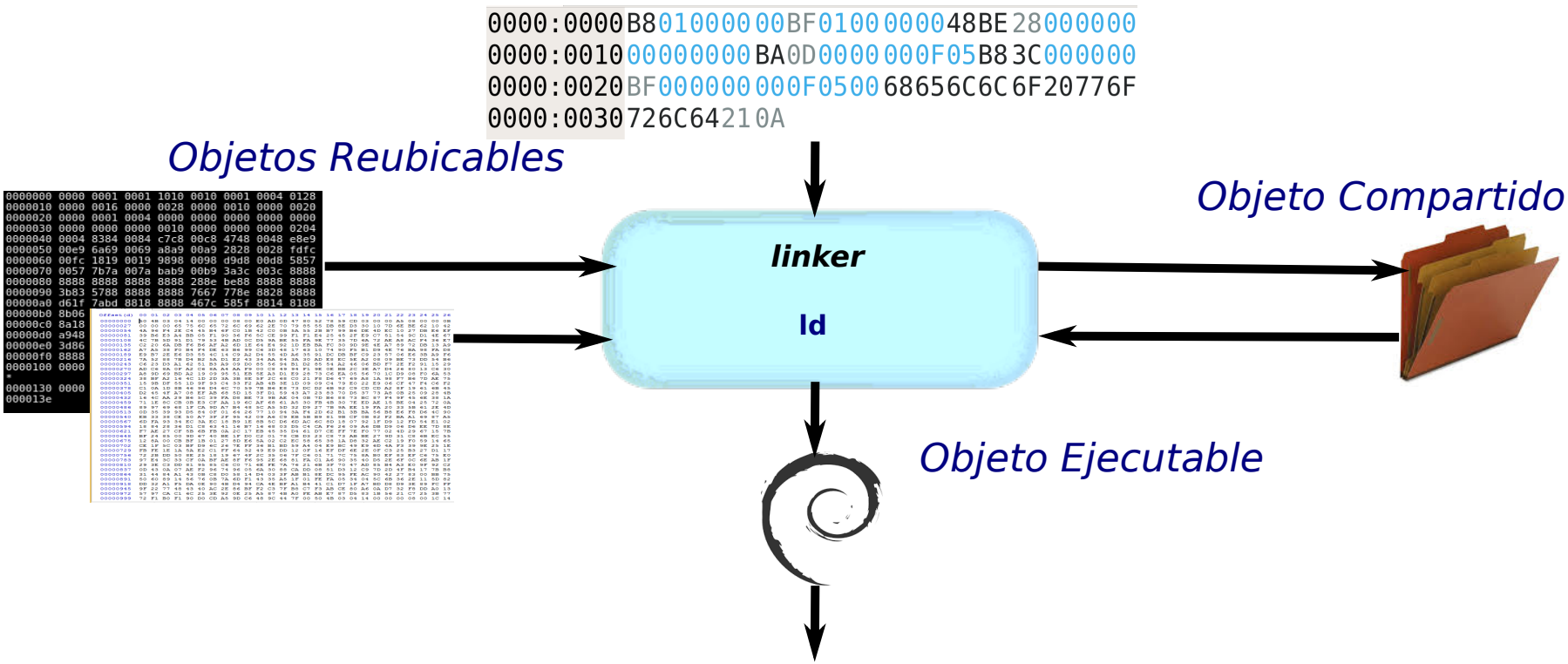
```
ld --verbose
```

Puede ser corroborado dentro del propio archivo /usr/bin/ld, ejecutando

```
hexdump -C /usr/bin/ld |more
```

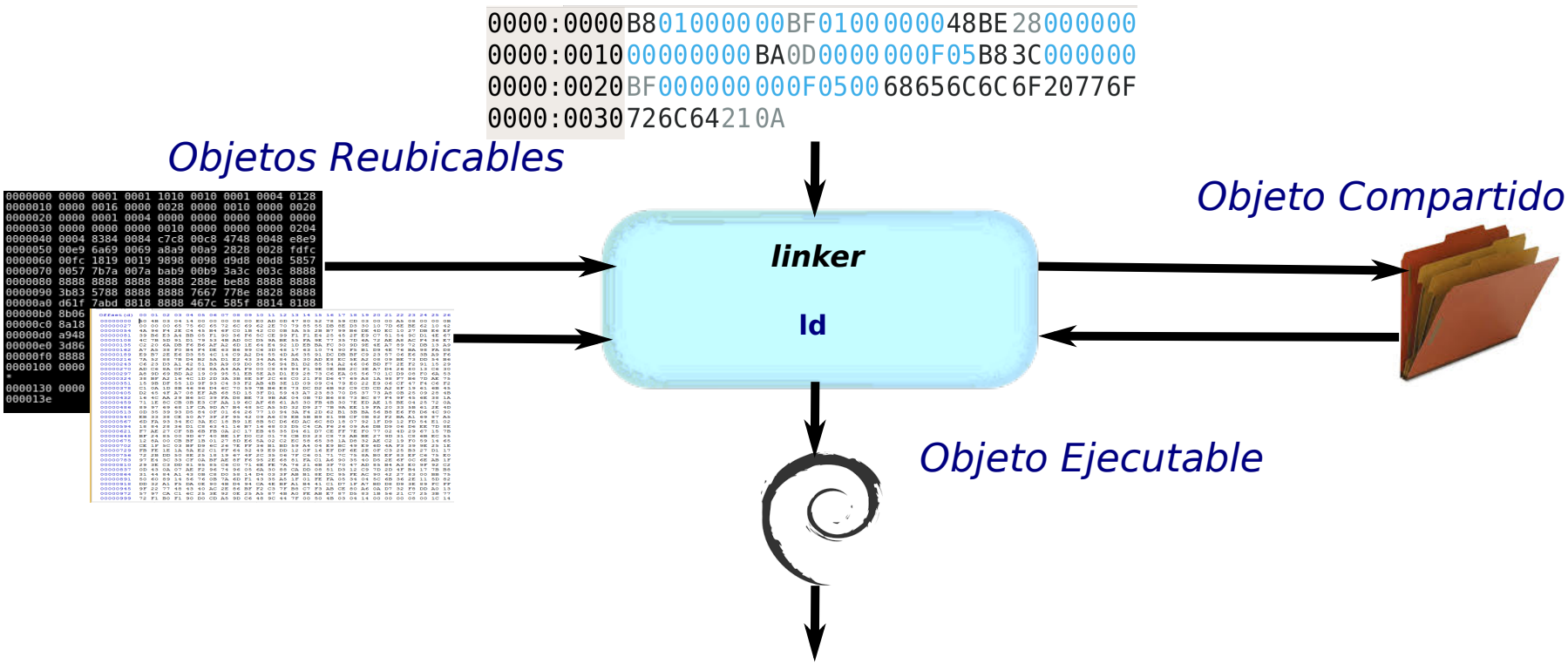
y navegando con paciencia a lo largo del código hasta encontrar la sección de datos en la que está escrito el script (por eso la opción **`-C`**)

Conceptos básicos



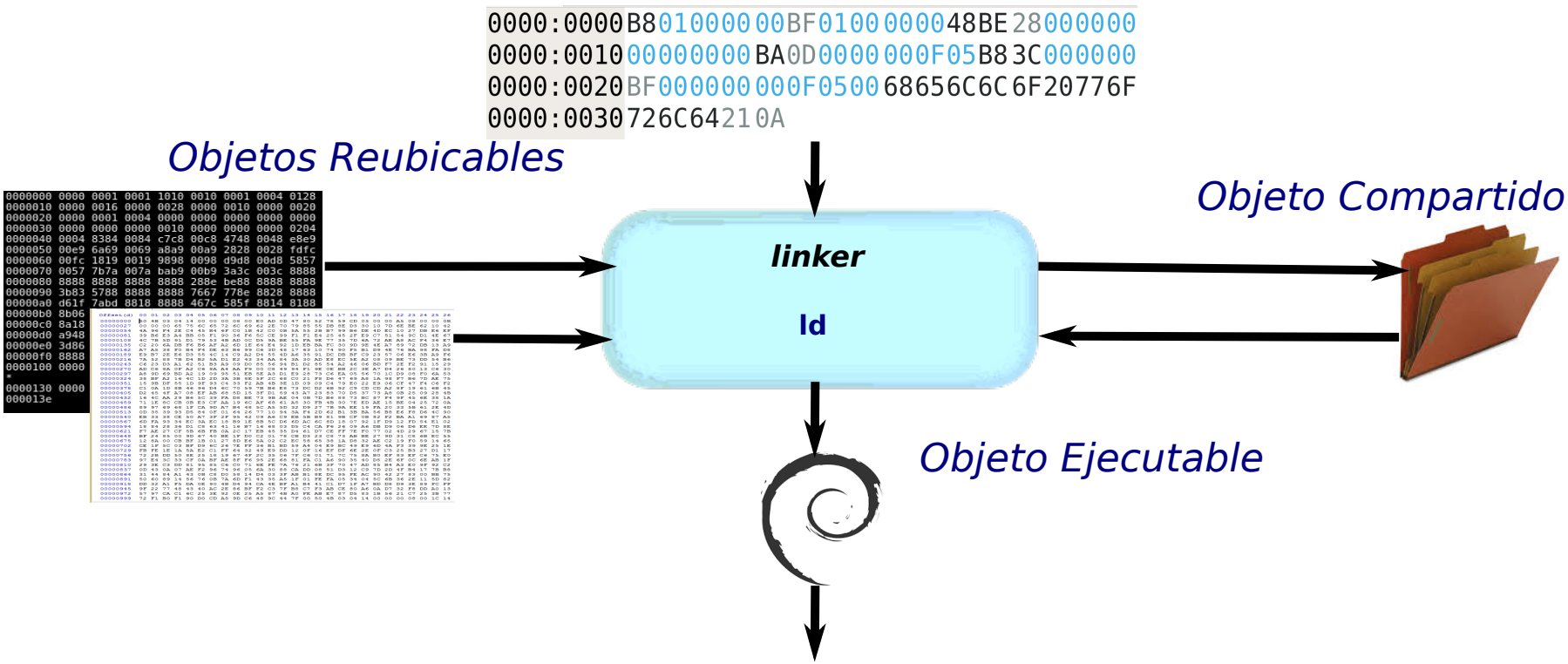
- Cada objeto reubicable está compuesto por secciones
- Lo mismo ocurre con el Objeto Ejecutable.
- El Linker básicamente combina secciones de entrada (es decir de los objetos reubicables) en secciones de salida (las que componen el objeto ejecutable

Conceptos básicos



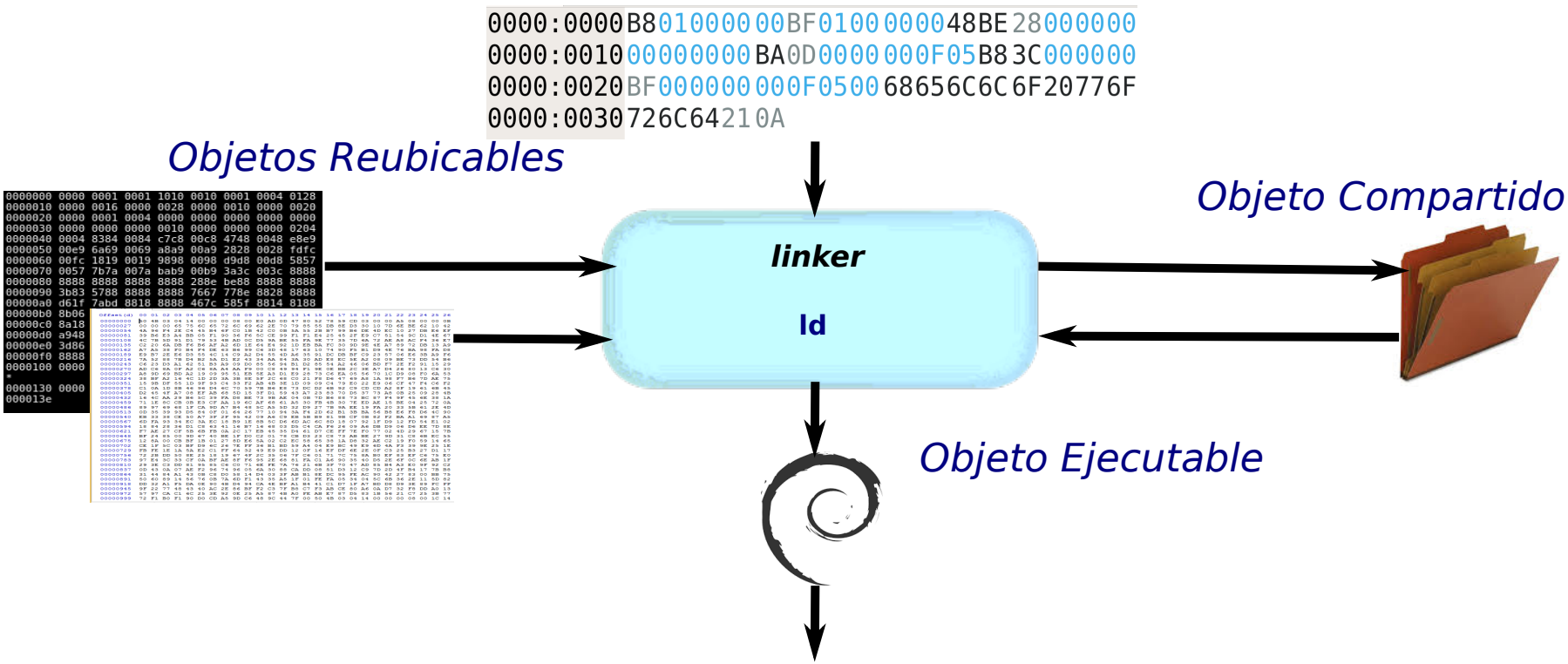
- Cada objeto reubicable está compuesto por secciones
- Lo mismo ocurre con el Objeto Ejecutable.
- El Linker básicamente combina secciones de entrada (es decir de los objetos reubicables) en secciones de salida (las que componen el objeto ejecutable

Conceptos básicos



- Cada objeto reubicable está compuesto por secciones
- Lo mismo ocurre con el Objeto Ejecutable.
- El Linker básicamente combina secciones de entrada (es decir de los objetos reubicables) en secciones de salida (las que componen el objeto ejecutable

Conceptos básicos



- Cada objeto reubicable está compuesto por secciones
- Lo mismo ocurre con el Objeto Ejecutable.
- El Linker básicamente combina secciones de entrada (es decir de los objetos reubicables) en secciones de salida (las que componen el objeto ejecutable

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser *loadable*, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser *allocatable*, lo que significa que si bien no tiene un contenido específico debe reservarse espacio en memoria para esta sección (eventualmente ser inicializada con ceros)
  - Si no es *allocatable* o *loadable*, la sección contiene únicamente información de depuración

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser *loadable*, lo que significa que su contenido se cargará en memoria al momento de la ejecución.
  - En otros casos la sección puede ser *allocatable*, lo que significa que si bien no tiene un contenido específico cuenta con reservar espacio en memoria para esta sección (eventualmente ser inicializada con ceros).
  - Si no es *allocatable* o *loadable*, la sección contiene únicamente información de depuración.

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser ***loadable***, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser ***allocatable***, lo que significa que si bien no tiene un contenido específico dentro de él, se reservará memoria para esta sección (eventualmente se inicializa con ceros)
  - Si no es *allocatable* o *loadable*, la sección simplemente contiene información de depuración

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser ***loadable***, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser ***allocatable***, lo que significa que si bien no tiene un contenido específico debe reservarse memoria para esta sección (eventualmente se inicializa con ceros)
  - Si no es ***allocatable*** o ***loadable***, la sección contiene típicamente información de debuging

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser ***loadable***, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser ***allocatable***, lo que significa que si bien no tiene un contenido específico debe reservarse memoria para esta sección (eventualmente se inicializa con ceros)
  - Si no es ***allocatable*** o ***loadable***, la sección contiene típicamente información de debuging

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser ***loadable***, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser ***allocatable***, lo que significa que si bien no tiene un contenido específico debe reservarse memoria para esta sección (eventualmente se inicializa con ceros)
  - Si no es ***allocatable*** o ***loadable***, la sección contiene típicamente información de debuging

Secciones

- Cada objeto (reubicable o ejecutable) está compuesto por secciones
- Cada sección posee un nombre, un tamaño, y un contenido (genéricamente un bloque de datos).
- Además las secciones poseen ciertos atributos
  - Una sección puede ser ***loadable***, lo que significa que su contenido se cargará en memoria al momento de la ejecución
  - En otros casos la sección puede ser ***allocatable***, lo que significa que si bien no tiene un contenido específico debe reservarse memoria para esta sección (eventualmente se inicializa con ceros)
  - Si no es ***allocatable*** o ***loadable***, la sección contiene típicamente información de debuging

Secciones *loadable* y *allocatable*

Las secciones de salida ***allocatables*** o ***loadables*** tienen dos tipos de direcciones:

1. **VMA**: Virtual Memory Address. Es la dirección que tiene la sección en el momento en que se ejecuta el archivo de salida. Esto es en run time.

2. **LMA**: Load Memory Address. Es la dirección en que la sección se cargará efectivamente en memoria.

Secciones *loadable* y *allocatable*

Las secciones de salida ***allocatables*** o ***loadables*** tienen dos tipos de direcciones:

1. **`VMA`**: Virtual Memory Address. Es la dirección que tiene la sección en el momento en que se ejecuta el archivo de salida. Esto es en run time.
2. **`LMA`**: Load Memory Address. Es la dirección en que la sección se cargará efectivamente en memoria.

Secciones *loadable* y *allocatable*

Las secciones de salida ***allocatables*** o ***loadables*** tienen dos tipos de direcciones:

1. **`VMA`**: Virtual Memory Address. Es la dirección que tiene la sección en el momento en que se ejecuta el archivo de salida. Esto es en run time.

2. **`LMA`**: Load Memory Address. Es la dirección en que la sección se cargará efectivamente en memoria.

Secciones *loadable* y *allocatable*

Las secciones de salida ***allocatables*** o ***loadables*** tienen dos tipos de direcciones:

1. **`VMA`**: Virtual Memory Address. Es la dirección que tiene la sección en el momento en que se ejecuta el archivo de salida. Esto es en run time.
2. **`LMA`**: Load Memory Address. Es la dirección en que la sección se cargará efectivamente en memoria.

Normalmente ambas direcciones son coincidentes. Pero justamente en el caso de desarrollo de firmware, cuyos objetos resultantes se almacenan en memorias ROM, estas dos direcciones no siempre coinciden. Tal es el caso de una sección de datos que se carga en ROM (utilizando la **`LMA`**) y luego para inicializar las variables se copia a RAM (utilizando la **`VMA`**).

Símbolos

- La lista de símbolos de un objeto puede ser vacía o puede contener información.
- Cada símbolo definido tiene, entre otra información, un nombre y una dirección.
- Si el símbolo pertenece a una función no definida en el archivo de entrada se marca U (Undefined), de otro modo puede ser local (l), global (g), o ambos (!).
- Para ver la tabla de símbolos tenemos varios comandos:

```
objdump -t archivo.objeto
nm archivo.objeto
```

Símbolos

- La lista de símbolos de un objeto puede ser vacía o puede contener información.
- Cada símbolo definido tiene, entre otra información, un nombre y una dirección.
- Si el símbolo pertenece a una función no definida en el archivo de entrada se marca U (Undefined), de otro modo puede ser local (l), global (g), o ambos (!).
- Para ver la tabla de símbolos tenemos varios comandos:

```
objdump -t archivo.objeto
nm archivo.objeto
```

Símbolos

- La lista de símbolos de un objeto puede ser vacía o puede contener información.
- Cada símbolo definido tiene, entre otra información, un nombre y una dirección.
- Si el símbolo pertenece a una función no definida en el archivo de entrada se marca U (Undefined), de otro modo puede ser local (l), global (g), o ambos (!).
- Para ver la tabla de símbolos tenemos varios comandos:

```
objdump -t archivo.objeto
nm archivo.objeto
```

Símbolos

- La lista de símbolos de un objeto puede ser vacía o puede contener información.
- Cada símbolo definido tiene, entre otra información, un nombre y una dirección.
- Si el símbolo pertenece a una función no definida en el archivo de entrada se marca U (Undefined), de otro modo puede ser local (l), global (g), o ambos (!).
- Para ver la tabla de símbolos tenemos varios comandos:

```
objdump -t archivo.objeto
nm archivo.objeto
```

Símbolos

- La lista de símbolos de un objeto puede ser vacía o puede contener información.
- Cada símbolo definido tiene, entre otra información, un nombre y una dirección.
- Si el símbolo pertenece a una función no definida en el archivo de entrada se marca U (Undefined), de otro modo puede ser local (l), global (g), o ambos (!).
- Para ver la tabla de símbolos tenemos varios comandos:

```
objdump -t archivo.objeto
nm archivo.objeto
```

Mirando símbolos (y entendiendo)

```
alejandro@DarkSideOfTheMoon:~/work/facu/TDIII/ProgramasClase/2019/mine/tp4$ objdump -t ../.
./../Assembler/ASM-Linux/2019/sleep1/sleep.o

../../../Assembler/ASM-Linux/2019/sleep1/sleep.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 sleep.asm
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l     O .data  0000000000000001 msg1
0000000000000010 l       *ABS*  0000000000000000 l1
0000000000000010 l     O .data  0000000000000001 msg2
000000000000001c l       *ABS*  0000000000000000 l2
000000000000002c l     O .data  0000000000000001 msg3
0000000000000034 l       *ABS*  0000000000000000 l3
0000000000000060 l     O .data  0000000000000004 timespec
0000000000000000 l     O .bss   0000000000000001 pid
0000000000000000 l    d  .debug_info    0000000000000000 .debug_info
0000000000000000 l    d  .debug_abbrev  0000000000000000 .debug_abbrev
0000000000000000 l    d  .debug_line    0000000000000000 .debug_line
0000000000000000         *UND*  0000000000000000 integer2BCD
0000000000000000         *UND*  0000000000000000 BCD2ascii
0000000000000000         *UND*  0000000000000000 memcopy
0000000000000000 g       .text  0000000000000000 _start
```

Mirando símbolos (y entendiendo)

```
alejandro@DarkSideOfTheMoon:~/work/facu/TDIII/ProgramasClase/2019/mine/tp4$ nm ../../../Ass
embler/ASM-Linux/2019/sleep1/sleep.o
                 U BCD2ascii
                 U integer2BCD
0000000000000010 a l1
000000000000001c a l2
0000000000000034 a l3
                 U memcopy
0000000000000000 d msg1
0000000000000010 d msg2
000000000000002c d msg3
0000000000000000 b pid
0000000000000000 T _start
0000000000000060 d timespec
```

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Elementos básicos de un Linker script

- El linker script es un archivo de texto que obedece a reglas sintácticas.
- Podemos poner comentarios. Para ello es como el C /* comment */
- Se escribe como una serie de comandos.
- Cada comando es un keyword seguido de un argumento o al que se le asigna un símbolo.
- Necesitamos volver a lo simple. Retomemos el primer ejemplo

Arranque en el reset vector

Vamos a reescribir el programa que toma el control en la dirección de reset en Modo Real en 0xFFFFFFF0 desde una ROM de 4Kbytes mapeada en 0xFFFFF000, pero en lugar de generar un binario con nasm incluir al linker y utilizar un linker script para manejar las posiciones de los diferentes elementos del programa en las direcciones de memoria apropiadas.

Arranque en el reset vector

```
1  ; Armamos una ROM de 4 KBytes
2  ; El procesador arranca en FFFF0 y en modo real, con
3  ; lo cual el mapa de memoria es de 1MB
4
5  code_size EQU end - reset
6  GLOBAL reset
7
8  USE16
9  section .data
10 times (4096-code_size) db 0x90
11 section .resetVector
12 reset:
13         cli
14         jmp reset
15 aqui:
16         hlt
17         jmp aqui
18 align 16
19 end:
```

Arranque en el reset vector

```
1  ; Armamos una ROM de 4 KBytes
2  ; El procesador arranca en FFFF0 y en modo real, con
3  ; lo cual el mapa de memoria es de 1MB
4
5  code_size EQU end - reset
6  GLOBAL reset
7
8  USE16
9  section .data
10 times (4096-code_size) db 0x90
11 section .resetVector
12 reset:
13         cli
14         jmp reset
15 aqui:
16         hlt
17         jmp aqui
18 align 16
19 end:
```

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es `SECTIONS`.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es `SECTIONS`.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es `SECTIONS`.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es `SECTIONS`.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es `SECTIONS`.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es **`SECTIONS`**.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Diferencias respecto del código antetrior

- El mas importante es que hemos omitido ORG 0xFF000 para establecer en que dirección de memoria iniciará nuestro código de arranque.
- Esta definición la hacemos en el script.
- La pregunta es: ¿Como?
- El lenguaje de scripting del linker resuelve éste y muchos otros asuntos mucho mas complejos.
- Uno de los comandos mas poderosos y comunes de usar en el lenguaje de scripting es **`SECTIONS`**.
- Con este comando definimos el mapeo de las diferentes secciones de los diferentes objetos de entrada en el objeto de salida

Primer linker script básico

- Sintaxis del comando **SECTIONS**:

```
SECTIONS
{
    Comandos de Seccion
    Comandos de Seccion
    .....
}
```

- Los comandos de sección son muy amplios pero en nuestro caso necesitamos definir cada sección de salida y como se mapean en ella la(s) sección(es) de entrada.
- De este modo los diferentes “Comandos de Sección”enunciados de manera genérica mas arriba se ocupan de esto.
- Además del mapeo entradas $->$ salida necesitamos especificar la dirección de inicio de la sección.

Primer linker script básico

- Sintaxis del comando **SECTIONS**:

```
SECTIONS
{
    Comandos de Seccion
    Comandos de Seccion
    .....
}
```

- Los comandos de sección son muy amplios pero en nuestro caso necesitamos definir cada sección de salida y como se mapean en ella la(s) sección(es) de entrada.
- De este modo los diferentes “Comandos de Sección”enunciados de manera genérica mas arriba se ocupan de esto.
- Además del mapeo entradas $->$ salida necesitamos especificar la dirección de inicio de la sección.

Primer linker script básico

- Sintaxis del comando **SECTIONS**:

```
SECTIONS
{
    Comandos de Seccion
    Comandos de Seccion
    .....
}
```

- Los comandos de sección son muy amplios pero en nuestro caso necesitamos definir cada sección de salida y como se mapean en ella la(s) sección(es) de entrada.
- De este modo los diferentes “Comandos de Sección”enunciados de manera genérica mas arriba se ocupan de esto.
- Además del mapeo entradas $->$ salida necesitamos especificar la dirección de inicio de la sección.

Primer linker script básico

- Sintaxis del comando **SECTIONS**:

```
SECTIONS
{
    Comandos de Seccion
    Comandos de Seccion
    .....
}
```

- Los comandos de sección son muy amplios pero en nuestro caso necesitamos definir cada sección de salida y como se mapean en ella la(s) sección(es) de entrada.
- De este modo los diferentes “Comandos de Sección”enunciados de manera genérica mas arriba se ocupan de esto.
- Además del mapeo entradas $->$ salida necesitamos especificar la dirección de inicio de la sección.

Primer linker script básico

```
SECTIONS
{
   . = 0xFF000;   /* '.' es el location counter.
                     Al inicio del comando SECTIONS su valor es 0.
                     Se incrementa con el tamaño de las secciones.*/
   .data : {*(.data)}
                     /*Formato:
                     Sección de salida : lista de las secciones de
                     entrada.
                     La lista es {archivo(sección)}. El '*' indica
                     cualquier archivo de entrada. Entre paréntesis
                     el nombre de la sección.
                     Esta línea colecta todas las secciones .data de
                     todos los archivos de entrada y las combina en
                     una unica sección .data en el archivo de salida
                         */
   . = 0xFFFF0;
   .text : {*(.resetVector)}
}
```

Como funciona (Detrás de escena)

- Para comenzar cambia radicalmente el ensamblado del archivo.
- Ya no generamos directamente el binario con **nasm**. Ahora generaremos un objeto reubicable **elf** de 32 bits (ver línea de comando **nasm** opción **-f** a continuación).

```
nasm -o mibios.elf -felf32 mibios.asm -l mibios.lst -Wall
```

- En la consola en donde termina de ensamblar el fuente minibios.asm, tipear

```
objdump -h -t mibios.elf
```

Como funciona (Detrás de escena)

- Para comenzar cambia radicalmente el ensamblado del archivo.
- Ya no generamos directamente el binario con **nasm**. Ahora generaremos un objeto reubicable **elf** de 32 bits (ver línea de comando **nasm** opción **-f** a continuación).

```
nasm -o mibios.elf -felf32 mibios.asm -l mibios.lst -Wall
```

- En la consola en donde termina de ensamblar el fuente minibios.asm, tipear

```
objdump -h -t mibios.elf
```

Como funciona (Detrás de escena)

- Para comenzar cambia radicalmente el ensamblado del archivo.
- Ya no generamos directamente el binario con **`nasm`**. Ahora generaremos un objeto reubicable **`elf`** de 32 bits (ver línea de comando **`nasm`** opción **`-f`** a continuación).

```
nasm -o mibios.elf -felf32 mibios.asm -l mibios.lst -Wall
```

- En la consola en donde termina de ensamblar el fuente minibios.asm, tipear

```
objdump -h -t mibios.elf
```

Como funciona (Detrás de escena)

- Para comenzar cambia radicalmente el ensamblado del archivo.
- Ya no generamos directamente el binario con **nasm**. Ahora generaremos un objeto reubicable **elf** de 32 bits (ver línea de comando **nasm** opción **-f** a continuación).

```
nasm -o mibios.elf -felf32 mibios.asm -l mibios.lst -Wall
```

- En la consola en donde termina de ensamblar el fuente minibios.asm, tipear

```
objdump -h -t mibios.elf
```

Así funciona... Es decir, el linker lee esto!

```
$ objdump -h -t mibios.elf

mibios.elf:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000ff0  00000000  00000000  00000130  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .resetVector  00000010  00000000  00000000  00001120  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 mibios.asm
00000000 l    d  .data  00000000 .data
00000000 l    d  .resetVector   00000000 .resetVector
00000010 l       *ABS*  00000000 code_size
00000000 l       .resetVector   00000000 reset
00000003 l       .resetVector   00000000 aqui
00000010 l       .resetVector   00000000 end
```

Invocando al linker

```
ld -z max-page-size=4096 -m elf_i386 -T mibios.ls -e reset -Map
    mibios.map --oformat=binary mibios.elf -o mibios.bin -Wall
```

- `-z` es un switch que permite enviar a `ld` palabras clave con su correspondiente valor. En nuestro caso lo utilizamos para especificar el tamaño de cada página de memoria en 4096 bytes (4 Kbyte).
- `-m` especifica una determinada emulación dentro de las soportadas por el linker. Para ver las diferentes emulaciones basta con tipear.

```
ld --verbose
```

y el resultado es al principio en el primer bloque de datos la lista de emulaciones soportadas: elf_x86_64, elf32_x86_64, elf_i386, elf_iamcu, i386linux, elf_l1om , elf_k1om, i386pep, i386pe

Invocando al linker

```
ld -z max-page-size=4096 -m elf_i386 -T mibios.ls -e reset -Map
    mibios.map --oformat=binary mibios.elf -o mibios.bin -Wall
```

- **`-z`** es un switch que permite enviar a **`ld`** palabras clave con su correspondiente valor. En nuestro caso lo utilizamos para especificar el tamaño de cada página de memoria en 4096 bytes (4 Kbyte).
- **`-m`** especifica una determinada emulación dentro de las soportadas por el linker. Para ver las diferentes emulaciones basta con tipear.

```
ld --verbose
```

y el resultado es al principio en el primer bloque de datos la lista de emulaciones soportadas: elf_x86_64, elf32_x86_64, elf_i386, elf_iamcu, i386linux, elf_l1om , elf_k1om, i386pep, i386pe

Invocando al linker

```
ld -z max-page-size=4096 -m elf_i386 -T mibios.ls -e reset -Map
    mibios.map --oformat=binary mibios.elf -o mibios.bin -Wall
```

- **-z** es un switch que permite enviar a **ld** palabras clave con su correspondiente valor. En nuestro caso lo utilizamos para especificar el tamaño de cada página de memoria en 4096 bytes (4 Kbyte).
- **-m** especifica una determinada emulación dentro de las soportadas por el linker. Para ver las diferentes emulaciones basta con tipear.

```
ld --verbose
```

y el resultado es al principio en el primer bloque de datos la lista de emulaciones soportadas: elf_x86_64, elf32_x86_64, elf_i386, elf_iamcu, i386linux, elf_l1om , elf_k1om, i386pep, i386pe

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- `-T` especifica que se pone un linker script para procesar con el linker.
- `-e` especifica el punto de entrada del objeto.
- `-Map` permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- `--oformat` Define el formato del archivo de salida. En este caso `--oformat=binary`, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- `-o` por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso `mibios.bin`.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico `-Wall`

Invocando al linker

- **`-T`** especifica que se pone un linker script para procesar con el linker.
- **`-e`** especifica el punto de entrada del objeto.
- **`-Map`** permite generar un archivo Map en el que podemos visualizar las secciones y sus características en el archivo de salida tal como han sido generadas por el linker.
- **`--oformat`** Define el formato del archivo de salida. En este caso **`--oformat=binary`**, establece como formato de salida el binario crudo que venimos utilizando como imagen de ROM.
- **`-o`** por *output*, sirve para especificar el nombre del archivo de salida. En nuestro caso **`mibios.bin`**.
- Como es habitual para tener el reporte completo de Warnings de modo de minimizar nuestros riesgos de bugs, utilizamos el clásico **`-Wall`**

Conclusiones hasta aqui

- Hicimos un ejemplo con lo mas básico de linker script.
- Nos permitió eliminar algunas directivas del programa fuente ya que en lugar de manejar las direcciones de inicio en el ensamblador lo hace el linker cuando se lo indicamos en el script de manera apropiada (ORG desapareció a expensas de definir la sección `ROM_init`).
- nasm debe generar como salida un ELF para que el Linker acceda a los símbolos y secciones definidas en el / los programas fuente.

Conclusiones hasta aqui

- Hicimos un ejemplo con lo mas básico de linker script.
- Nos permitió eliminar algunas directivas del programa fuente ya que en lugar de manejar las direcciones de inicio en el ensamblador lo hace el linker cuando se lo indicamos en el script de manera apropiada (ORG desapareció a expensas de definir la sección `ROM_init`).
- nasm debe generar como salida un ELF para que el Linker acceda a los símbolos y secciones definidas en el / los programas fuente.

Conclusiones hasta aqui

- Hicimos un ejemplo con lo mas básico de linker script.
- Nos permitió eliminar algunas directivas del programa fuente ya que en lugar de manejar las direcciones de inicio en el ensamblador lo hace el linker cuando se lo indicamos en el script de manera apropiada (ORG desapareció a expensas de definir la sección **`ROM_init`**).
- nasm debe generar como salida un ELF para que el Linker acceda a los símbolos y secciones definidas en el / los programas fuente.

Conclusiones hasta aqui

- Hicimos un ejemplo con lo mas básico de linker script.
- Nos permitió eliminar algunas directivas del programa fuente ya que en lugar de manejar las direcciones de inicio en el ensamblador lo hace el linker cuando se lo indicamos en el script de manera apropiada (ORG desapareció a expensas de definir la sección **`ROM_init`**).
- nasm debe generar como salida un ELF para que el Linker acceda a los símbolos y secciones definidas en el / los programas fuente.

Definiendo Secciones de Salida

- Nuestro simpático “ls”del apartado anterior ha sido, digamos, minimalista.
- Cumplido el objetivo de romper el hielo, vamos a darle alguna funcionalidad adicional para ir explorando las múltiples capacidades que tenemos a medida que avanzamos en el dominio de su sintaxis y los conceptos asociados.
- Un área a explorar es la que define las secciones de salida (es decir las del objeto ejecutable que construirá el linker. Su descripción completa es así:

Definiendo Secciones de Salida

- Nuestro simpático “ls”del apartado anterior ha sido, digamos, minimalista.
- Cumplido el objetivo de romper el hielo, vamos a darle alguna funcionalidad adicional para ir explorando las múltiples capacidades que tenemos a medida que avanzamos en el dominio de su sintaxis y los conceptos asociados.
- Un área a explorar es la que define las secciones de salida (es decir las del objeto ejecutable que construirá el linker. Su descripción completa es así:

Definiendo Secciones de Salida

- Nuestro simpático “ls”del apartado anterior ha sido, digamos, minimalista.
- Cumplido el objetivo de romper el hielo, vamos a darle alguna funcionalidad adicional para ir explorando las múltiples capacidades que tenemos a medida que avanzamos en el dominio de su sintaxis y los conceptos asociados.
- Un área a explorar es la que define las secciones de salida (es decir las del objeto ejecutable que construirá el linker. Su descripción completa es así:

Definiendo Secciones de Salida

- Nuestro simpático “ls”del apartado anterior ha sido, digamos, minimalista.
- Cumplido el objetivo de romper el hielo, vamos a darle alguna funcionalidad adicional para ir explorando las múltiples capacidades que tenemos a medida que avanzamos en el dominio de su sintaxis y los conceptos asociados.
- Un área a explorar es la que define las secciones de salida (es decir las del objeto ejecutable que construirá el linker. Su descripción completa es así:

Definiendo Secciones de Salida

- Nuestro simpático “ls”del apartado anterior ha sido, digamos, minimalista.
- Cumplido el objetivo de romper el hielo, vamos a darle alguna funcionalidad adicional para ir explorando las múltiples capacidades que tenemos a medida que avanzamos en el dominio de su sintaxis y los conceptos asociados.
- Un área a explorar es la que define las secciones de salida (es decir las del objeto ejecutable que construirá el linker. Su descripción completa es así:

```
section [address ] [(type )] : [AT(lma )]
{
    output-section-command
    output-section-command
...
} [>region ] [AT>lma_region ] [:phdr :phdr ...] [=fillexp ]
```

Definiendo Secciones de Salida

Hasta ahora la definición de nuestra sección de salida se ha reducido simplemente a:

```
{
...
    .text : {*(.resetVector)}
...
}
```

- Lo cual demuestra que no siempre se usan todas las posibilidades.
- Nuevo concepto: el Location Counter (o sea nuestro modesto operador '.') se refiere a Virtual Memory Adress (**VMA**)

Definiendo Secciones de Salida

Hasta ahora la definición de nuestra sección de salida se ha reducido simplemente a:

```
{
...
    .text : {*(.resetVector)}
...
}
```

- Lo cual demuestra que no siempre se usan todas las posibilidades.
- Nuevo concepto: el Location Counter (o sea nuestro modesto operador '.') se refiere a Virtual Memory Adress (VMA)

Definiendo Secciones de Salida

Hasta ahora la definición de nuestra sección de salida se ha reducido simplemente a:

```
{
...
    .text : {*(.resetVector)}
...
}
```

- Lo cual demuestra que no siempre se usan todas las posibilidades.
- Nuevo concepto: el Location Counter (o sea nuestro modesto operador '.') se refiere a Virtual Memory Adress (**VMA**)

Definiendo Secciones de Salida

Hasta ahora la definición de nuestra sección de salida se ha reducido simplemente a:

```
{
...
    .text : {*(.resetVector)}
...
}
```

- Lo cual demuestra que no siempre se usan todas las posibilidades.
- Nuevo concepto: el Location Counter (o sea nuestro modesto operador '.') se refiere a Virtual Memory Adress (**VMA**)

```
...
    .=0xFFFFFFF0
...
```

Se refiere a una dirección Virtual, es decir, en donde se terminará ejecutando el código. Si no especificamos otra cosa el linker utiliza **LMA=VMA**.

Definiendo Secciones de Salida

Vamos a mejorar el programa anterior copiándolo en una dirección de RAM apropiada para arrancar un S.O. Por ejemplo 0x7C00. Allí deberá transferir el control para pasar a Modo Protegido por ejemplo y terminar la inicialización.

Agregando secciones: En el código

```
USE16
GLOBAL  Entry
section .resetVector
Entry:
        cli
        jmp init
aqui:
        hlt
        jmp aqui
align 16
resetVector_size EQU $ - Entry

section .ROM_init
init_bootstrap:
idle:
        hlt
        jmp     idle
init:
;* Se activa la DRAM y su controlador
        DRAM_Enable     ;Macro definida en prox. slide
;* Seteamos un stack
        mov     sp,0x3000
```

Agregando secciones: En el código

```
        mov     ax,0
        mov     es,ax
        mov     di,0x7c00
        push    cs
        pop     ds
        mov     si,init_bootstrap
        mov     cx,init_size
        call    memcopy
        jmp     0x0:0x7C00

memcopy:
next:
        mov     al,byte[cs:si]  ; ver commentario al final
        stosb                   ;[es:di]<-al , di++
        inc     si              ;apuntamos al siguiente byte en la ROM
        loop    next            ;cx--, if(FLAGS.ZF==0) goto next
        ret
init_size EQU $ - init_bootstrap

%macro  DRAM_Enable 0  ;Macro. Formato % macro macro_name #argumentos
        nop
%endmacro
```

Agregando secciones: En el Linker Script

```
SECTIONS
{
      . = 0xFFFFFFF0; /*'.' es el location counter.
                  Al inicio del comando SECTIONS su valor es 0.
                  Se incrementa con el tama o de las secciones.
                  Su valor corresponde a Direcciones Virtuales*/
      .resetVector : AT ( 0xFFFFFFF0 ) {*(.resetVector)}

      . = 0xFFFFF000;
      .ROM_init : AT ( 0xFFFFF000 ) {*(.ROM_init)}
}
```

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea `A20`) que simplemente no existía en el 8086. El resultado es `0xFFFF:0xFFFF = 0x0FFEF`.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea **A20**) que simplemente no existía en el 8086. El resultado es **0xFFFF:0xFFFF = 0x0FFEF**.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea `A20`) que simplemente no existía en el 8086. El resultado es `0xFFFF:0xFFFF = 0x0FFEF`.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea **`A20`**) que simplemente no existía en el 8086. El resultado es **`0xFFFF:0xFFFF = 0x0FFEF`**.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea **`A20`**) que simplemente no existía en el 8086. El resultado es **`0xFFFF:0xFFFF = 0x0FFEF`**.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea **`A20`**) que simplemente no existía en el 8086. El resultado es **`0xFFFF:0xFFFF = 0x0FFEF`**.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

Problemática en Modo Real

- El 8086 restringido a 1Mbyte de Memoria, administrada por segmentos cuyo tamaño máximo era 64Kbytes, y soporte de *wrap around*, generó prácticas de programación algo desprolijas.
- Una dirección lógica 0xFFFF:0xFFFF, debido a la propiedad conocida como *wrap around* de los segmentos, daba como resultado la siguiente dirección física:

$$0xFFFF << 4 + 0xFFFF = 0x10FFEF$$

- El 1 mas significativo se descarta debido a que corresponde al bit 20 del bus de address (línea **`A20`**) que simplemente no existía en el 8086. El resultado es **`0xFFFF:0xFFFF = 0x0FFEF`**.
- Esta propiedad (sin entrar en valoraciones subjetivas acerca de su conveniencia) fue aprovechada por no pocos programadores.
- Los procesadores siguientes inician su operación en el denominado Modo Real a fin de ser compatibles.
- Por lo tanto se necesitó respetar este comportamiento. Pero todos podían acceder muy por encima del Mbyte.

El 80286 direccionaba 16Mbytes. ¿Entonces?

- En el ejemplo anterior A20 enviaba un 1 hacia el bus de address.
- Este no es el comportamiento esperado por un 8086. O dicho de otro modo por un programa que trabaja en Modo Real, que como dijimos especularon con que **A20** nunca sería '1'.
- Como al lanzar el primer procesador que tenía capacidad de direccionamiento por encima del Mbyte, el 80286, Intel no hizo nada al respecto, fueron los fabricantes de MotherBoards (IBM particularmente) quienes debieron lidiar con la situación

El 80286 direccionaba 16Mbytes. ¿Entonces?

- En el ejemplo anterior A20 enviaba un 1 hacia el bus de address.
- Este no es el comportamiento esperado por un 8086. O dicho de otro modo por un programa que trabaja en Modo Real, que como dijimos especularon con que `A20` nunca sería '1'.
- Como al lanzar el primer procesador que tenía capacidad de direccionamiento por encima del Mbyte, el 80286, Intel no hizo nada al respecto, fueron los fabricantes de MotherBoards (IBM particularmente) quienes debieron lidiar con la situación

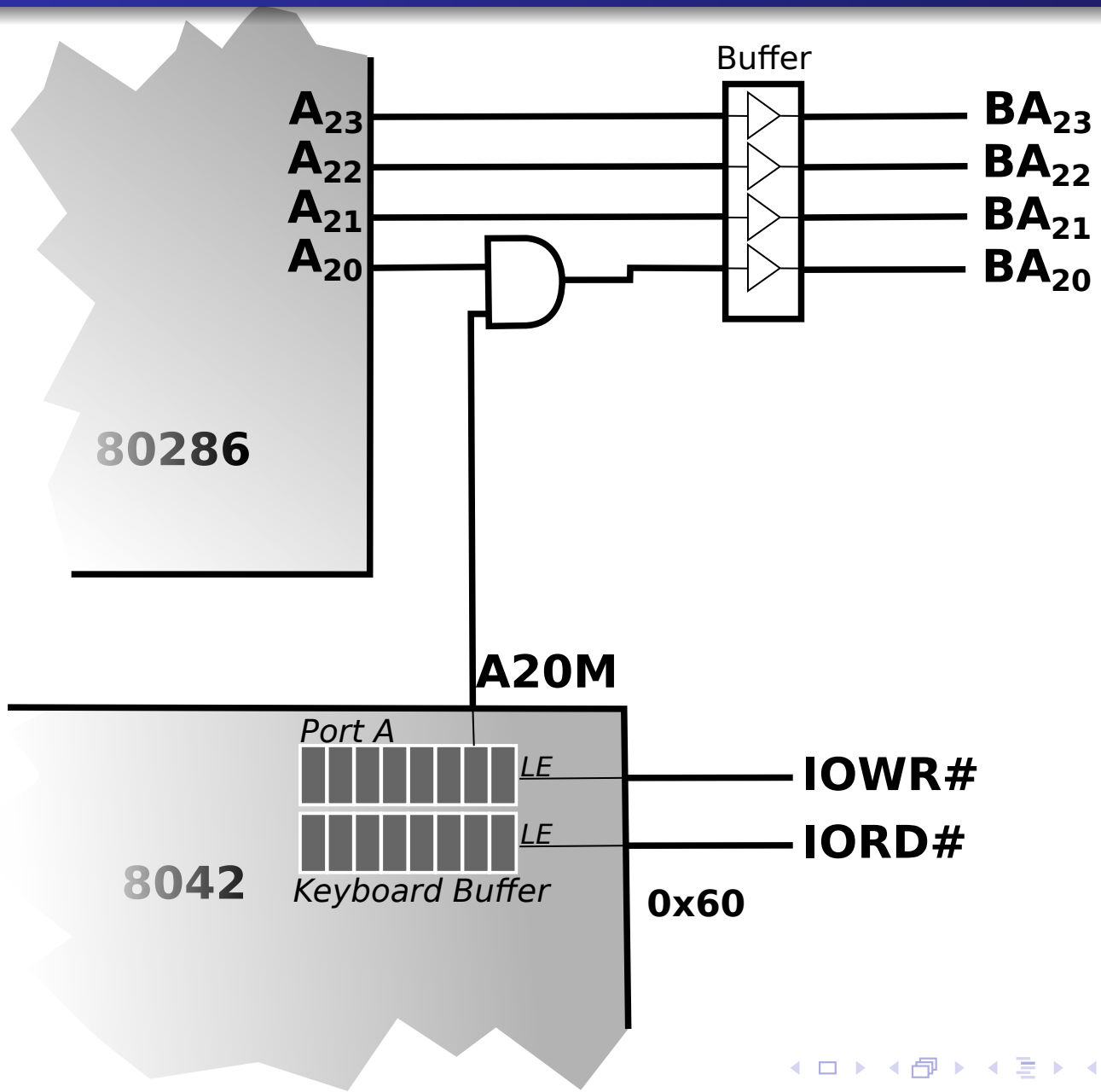
El 80286 direccionaba 16Mbytes. ¿Entonces?

- En el ejemplo anterior A20 enviaba un 1 hacia el bus de address.
- Este no es el comportamiento esperado por un 8086. O dicho de otro modo por un programa que trabaja en Modo Real, que como dijimos especularon con que **`A20`** nunca sería '1'.
- Como al lanzar el primer procesador que tenía capacidad de direccionamiento por encima del Mbyte, el 80286, Intel no hizo nada al respecto, fueron los fabricantes de MotherBoards (IBM particularmente) quienes debieron lidiar con la situación

El 80286 direccionaba 16Mbytes. ¿Entonces?

- En el ejemplo anterior A20 enviaba un 1 hacia el bus de address.
- Este no es el comportamiento esperado por un 8086. O dicho de otro modo por un programa que trabaja en Modo Real, que como dijimos especularon con que **`A20`** nunca sería '1'.
- Como al lanzar el primer procesador que tenía capacidad de direccionamiento por encima del Mbyte, el 80286, Intel no hizo nada al respecto, fueron los fabricantes de MotherBoards (IBM particularmente) quienes debieron lidiar con la situación

Hardware


Buffer
A23
A22
A21
A20
BA23
BA22
BA21
BA20
80286
A20M
Port A
LE
LE
IOWR#
IORD#
8042
Keyboard Buffer
0x60

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1'en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1'en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1' en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1'en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1'en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Funcionamiento

- Se activa o desactiva mediante un bit en el puerto de E/S 0x60.
- El port 0x60 corresponde al controlador 8042 cuya función principal es proveer los códigos de las teclas pulsadas en el teclado.
- Leer el port 0x60 permite obtener el código de la tecla recientemente pulsada.
- Escribir el port 0x60 permite establecer diversos comportamientos. En particular el bit de orden 1 es el encargado de habilitar o no la compuerta AND.
- Un '1'en ese bit permite a la compuerta AND colocar en su salida el estado de la línea A20 del procesador.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Soporte on chip

- Intel a partir del 80486 al incluir en el mismo chip, la CPU, el controlador cache y el cache level 1, incluye un terminal **A20M#**, que si se activa habilita el comportamiento compatible con el 8086.
- En el reset la línea A20 está habilitada (de otro modo no podría acceder a la dirección del reset vector en 0xFFFFFFF0, ya que el la línea A20 no podría estar en 1, lo que haría imposible sacar esa dirección al address bus.
- Para activar la función es necesario escribir '0' en el terminal del procesador **A20M#**.
- Siempre se hace en el bit 1 del port 0x60.
- Los fabricantes en el BIOS le activan un 0 en **A20M#** para que esté en modo 8086. Si el sistema operativo trabajará en modo real no debe hacer nada, pero si va a pasar a modo protegido deberá activar la línea **A20M#**.
- Desde la microarquitectura Haswell Intel ya no da soporte a esta funcionalidad.

Código

Nos proponemos agregar código de A20 y dejarlo listo para entrar a a Modo Protegido.
Luego un segundo round de Linker script

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Armando un mapa de memoria

- Por default el linker permite acceder a toda la memoria disponible.
- Sin embargo provee herramientas para que podamos personalizar el uso de memoria y las propiedades de cada área.
- El comando es **MEMORY**, que permite definir la ubicación de los bloques de memoria y su tamaño en el archivo de salida.
- Sintaxis del comando **MEMORY**:

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    .....
}
```

- En nuestro caso escribamos:

```
MEMORY
{
        RAM (!rx): ORIGIN = 0, LENGTH = 0xFFFFF000
        ROM (rx) : ORIGIN = 0XFFFFF000, LENGTH = 4K
}
```

Atributos de memoria

Los bloques de memoria que podemos definir deben tener diferentes atributos de acuerdo con las características de cada bloque. Los posibles atributos de explican a continuación

- ‘R’ Read-only section
- ‘W’ Read/write section
- ‘X’ Executable section
- ‘A’ Allocatable section
- ‘I’ Initialized section
- ‘L’ Idem ’I’
- ‘!’ Invierte el sentido de todos los atributos precedentes